Escuela Superior
de Ingeniería

ESCUELA SUPERIOR DE INGENIERÍA

GRADO EN INGENIERÍA INFORMÁTICA

SISTEMA DE RASTREO DE TUITS EN TIEMPO REAL

Antonio Jesús Rodas Rueda

San Fernando, Febrero 2017

Escuela Superior
de Ingeniería

ESCUELA SUPERIOR DE INGENIERÍA

GRADO EN INGENIERÍA INFORMÁTICA

SISTEMA DE RASTREO DE TUITS EN TIEMPO REAL

Autor: Antonio Jesús Rodas Rueda
Director: Juan Boubeta-Puig
Directora: Guadalupe Ortiz Bellot

San Fernando, Febrero 2017

Agradecimientos

A Juan Boutbeta-Puig, por haber sido mi tutor y haberme proporcionado la orientación necesaria para empezar a desarrollar este TFG.

A Guadalupe Ortiz Bellot, por haber sido mi tutora y por proporcionarme su ayuda en todo momento en el que se han presentado dudas o se ha necesitado orientación sobre cualquier tema relacionado con el TFG.

A mi familia y amigos, por estar en los momentos difíciles, ayudándome a realizar este TFG mostrándome su apoyo y su confianza acerca de las posibilidades del proyecto.

Índice general

Índice de figuras

Índice de tablas

1. Introducción

A continuación, se exponen diferentes aspectos relacionados con este proyecto como son la motivación que ha provocado la elaboración del mismo, el tema que abarca y los objetivos que se quieren cumplir, entre otros.

1.1. Ámbito

Hoy en día, el mundo está conectado mediante Internet, y dentro de Internet, el medio que más información puede crear, modificar, exponer o compartir datos, son las redes sociales como pueden ser **Instagram**, **Facebook** o **Twitter**.

Una de las redes sociales más utilizadas, con más de 300 millones de usuarios activos, es **Twitter**.

Twitter, consiste en que cada usuario registrado en la plataforma puede describir en mensajes, de 140 caracteres como máximo, lo que se le venga a la mente en ese instante, cosas como: deseos, quejas, planes, gustos, opiniones, experiencias, colgar fotos, videos, compartir enlaces de noticias o dar consejos. Estos mensajes quedan registrados en la web con el fin de que los demás usuarios lo vean y a la vez puedan opinar sobre ello, apoyarlo, o simplemente leerlo porque no tenga nada mejor que hacer.

Esto conlleva a que en Twitter se genera y comparte una cantidad enorme de información que, procesada correctamente, es capaz de revelar en qué está pensando el mundo en este instante, qué es lo que les falta, qué es lo que les sobra, qué es lo que gusta o simplemente cuál es el tema del que todo el mundo habla en ese momento [3].

Este proyecto consiste en usar **Twitter** y **Apache Storm**, un sistema de computación en tiempo real, que permite procesar esa información en tiempo real para así poder gestionarla y usarla con el fin de realizar estudios de mercado, estudios estadísticos, publicidad dirigida o simplemente saber de qué se está hablando en un lugar concreto en ese momento.

1.2. Motivación

La motivación principal de este proyecto es el aprendizaje de nuevas tecnologías que se encuentran a la vanguardia en cuanto al desarrollo de aplicaciones web se refiere.

Además se pretende que este proyecto sea algo que no va a quedar en el olvido ya que las redes sociales son algo que todo el mundo utiliza y que tiene un largo ciclo de vida. Los usuarios de estas plataformas son cada vez más y son más utilizadas como sistema de comunicación puesto que cada vez es más habitual ver cómo programas de televisión, radio o incluso informativos, utilizan este medio para comunicarse con sus espectadores,

buscando la participación de los mismos en el programa o estudiando las opiniones que se generan para ver si se está realizando un buen trabajo o hay cosas que mejorar.

Lo que se quiere dar a entender es que las redes sociales, lejos de estar en decadencia o ser una moda pasajera, se están convirtiendo en el sistema que mueva más información en la actualidad y por tanto, en la mayor fuente de conocimiento.

1.3. Objetivos

El objetivo principal de este proyecto es crear un sistema de procesamiento de una gran cantidad de información producida en tiempo real. Un sistema capaz de generar tanta información en poco espacio de tiempo es Twitter.

Además, se pretende que el proceso de obtener la información sea parametrizable, consiguiendo así que cada usuario de la aplicación pueda obtener unos datos personalizados en cada caso.

Por tanto los subobjetivos son:

- Crear un sistema que sea capaz de procesar todos *los **tuits***, que se escriban en Twitter, en España, durante un periodo de tiempo determinado.

- El sistema será capaz de capturar los *tuits* durante un **tiempo determinado** que podrá especificar el usuario.

- Será posible **parametrizar** la búsqueda de *tuits* mediante **palabras** o **conjunto de palabras** para obtener los datos más personalizados posibles, de forma que el usuario indicará una palabra, frase o un conjunto de palabras separadas y el sistema capturará los *tuits* que contengan esos parámetros.

- Además, el usuario podrá establecer el nivel de **detalle geográfico** por el que se mostrarán los *tuits*. Será posible mostrar los *tuits* a nivel **regional** o a nivel **provincial.**

- Los *tuits* obtenidos se representarán gráficamente en un mapa dividido en regiones dependiendo del detalle geográfico escogido por el usuario (regional o provincial), coloreando las regiones donde se hayan escrito un *tuit* en ese instante. El color de la región variará dependiendo del número de *tuits* que se hayan obtenido en ese lugar.

- La aplicación dispondrá de un diseño *responsive* para que pueda ser utilizada en dispositivos móviles con una pantalla de menor tamaño.

1.4. Alcance

Este proyecto, es una aplicación web, la cual podrá ser usada por cualquier individuo particular, institución pública o empresa que pretenda obtener información acerca de donde es tendencia un tema particular en el cual estén interesados.

La infraestructura necesaria para el despliegue de la aplicación es simplemente un servidor. Y para acceder a la aplicación solo se necesita acceso a una conexión de Internet.

1.5. Visión general

En los próximos capítulos se hablará del estado del arte de las diversas tecnologías usadas para la realización de la aplicación, la causa y el propósito para el que se han usado.

Se detallará el proceso de desarrollo del proyecto explicando la planificación y las fases por las que ha pasado el mismo mediante la exposición de un diagrama de Gantt.

A continuación, se procederá a realizar la descripción de las diferentes características del proyecto en la cual se explicarán las diferentes interfaces de las que consta junto el funcionamiento de cada una de ellas.

En capítulos posteriores, se hablará del proceso de desarrollo del proyecto en un aspecto más técnico donde se verá en qué consisten sus requisitos, diseño, implementación y pruebas realizadas.

Al final del documento se realizará un resumen del proyecto, junto con las conclusiones y las valoraciones que se han obtenido al desarrollar esta aplicación.

1.6. Glosario

1.6.1. Definiciones

- **Apache Storm:** Sistema de procesamiento de datos en tiempo real.
- **Clojure: L**enguaje de programación de propósito general dialecto de Lisp.

- **Clúster:** Conjunto de ordenadores unidos entre sí mediante una red y que se comportan como si fueran una sola máquina.

- **Framework:** Conjunto estandarizado de conceptos prácticas y criterios que sirven para resolver tipos problemas de similares características.

- **Hadoop:** Framework de software que soporta aplicaciones distribuidas [6].

- **Lisp:** Familia de lenguajes de programación de computadora de tipo multiparadigma
- **Localhost:** Es un nombre reservado que tienen todas las computadoras para hacerse referencia a sí mismo en un ambiente de red.
- **Login:** Acción de iniciar sesión en un sistema.
- **Servlet:** Clase en lenguaje de programación Java, utilizada para ampliar las capacidades de un servidor.
- **Stateless:** Protocolo de comunicaciones sin estado [7].
- **Stream:** Sucesión de tuplas desordenadas.
- **Tuit:** Texto escrito en la red social Twitter, que consta de 140 caracteres como máximo, en el que se pueden insertar imágenes, videos o enlaces.
- **Tupla:** Lista ordenada de elementos de cualquier tipo.

1.6.2. Acrónimos

- **API:** Application Programming Interface - Interfaz de Programación de Aplicaciones
- **CSS:** Cascading Style Sheets - Hojas de estilos en cascada
- **DAO:** Data Access Object - Objeto de acceso a datos
- **HTML:** HyperText Markup Language - Lenguaje de formato de documentos para hipertexto
- **POM:** Project Object Model – Modelo de datos del proyecto
- **SQL:** Structured Query Language - Lenguaje de consulta estructurado
- **TFG:** Trabajo de Fin de Grado
- **UCA:** Universidad de Cádiz
- **URL:** Uniform Resource Locator - Localizador Uniforme de Recursos
- **WAR:** Web Application Archive - Archivo de aplicación web
- **XML:** eXtensible Markup Language - Lenguaje de marcado extensible

2. Estado del arte

A continuación se describirán las diferentes tecnologías usadas para realizar el desarrollo de esta aplicación.

2.1. Apache Storm

Storm fue creado originalmente por **Nathan Marz** [4] y el **BackType**. BackType es una empresa de medición y analítica social que ha sido comprada por Twitter y que actualmente proporciona los datos necesarios para el funcionamiento de **Twitter Analytics** [5] entre otras aplicaciones.

Más tarde, Storm también fue adquirido y establecido de **código abierto** por Twitter. En poco tiempo, Apache Storm se convirtió en un estándar para los sistemas distribuidos de procesamiento de datos en tiempo real que nos permite procesar una gran cantidad de datos, algo similar a lo que hace Hadoop [6].

Apache Storm está desarrollado en Java y Clojure [14] y continúa siendo líder en el análisis en tiempo real.

2.1.1. ¿Qué es?

Storm es un sistema distribuido de procesamiento de datos en tiempo real. Está diseñado para procesar enormes cantidades de datos en un método tolerante a fallos y de forma escalable.

Aunque Storm es "*stateless*" [7], gestiona el entorno distribuido y el estado del *clúster* a través de *Apache Zookeeper* (explicado en el siguiente apartado).

Storm garantiza que cada mensaje será procesado a través de la topología al menos una vez.

2.1.2. Apache Storm vs Hadoop

Básicamente Hadoop y Storm, son frameworks que son utilizados para el análisis de Big Data. En la siguiente tabla se realiza una comparación entre ambos:

Storm	Hadoop
Procesamiento de datos en tiempo real	Procesamiento por lotes
Protocolo sin estado	Protocolo con estado
Arquitectura maestro (*Nimbus*) / esclavo (supervisor) coordinada mediante *Zookeeper*	Arquitectura maestro (*job tracker*) / esclavo (*task tracker*) coordinada mediante *Zookeeper*
A un proceso de *streaming* de Storm pueden procesar decenas de miles de mensajes por segundo en el *clúster*.	*Hadoop Distributed File System* (HDFS) utiliza el *framework MapReduce* para procesar gran cantidad de datos que tardan minutos u horas.
La topología de Storm puede ejecutarse indefinidamente hasta que el usuario la detenga o se produzca un fallo.	*MapReduce* se ejecuta en orden secuencial y terminan su ejecución.
Ambos son distribuidos y tolerantes a fallos	
Si *Nimbus* o Supervisor fallan, se puede reiniciar y seguir la ejecución desde el momento en el que falló.	Si *JobTracker* falla, se perderá la ejecución y tendrá que empezar de cero.

Tabla 1: Apache Storm vs Hadoop

2.1.3. Beneficios de Apache Storm

A continuación se muestra una lista de los beneficios que nos proporciona el uso de Apache Storm frente a otros sistemas de procesamiento de datos:

- Es de código abierto, estable, y de uso relativamente sencillo.
- Es tolerante a fallos, flexible, seguro y soporta todo tipo de lenguajes de programación.
- Permite procesamiento de datos en tiempo real.
- Tiene una gran capacidad de procesar datos, lo que hace que sea extremadamente rápido.
- Es altamente escalable, lo que provoca que la carga de trabajo pueda variar sin que el rendimiento se vea afectado a causa de la adición o eliminación de recursos.
- Tiene muy baja latencia por lo que es capaz de refrescar los datos rápidamente.
- Garantiza el procesamiento de los datos incluso si se pierde cualquiera de los nodos conectados en el *clúster*.

2.1.4. Conceptos

En esta sección se explicará el proceso por el que pasa la información que es tratada por *Storm*, además, se detallarán los componentes con los que cuenta el sistema para realizar el proceso.

En la ilustración 1 se muestra el proceso general por el que pasa la información junto con los componentes del sistema:

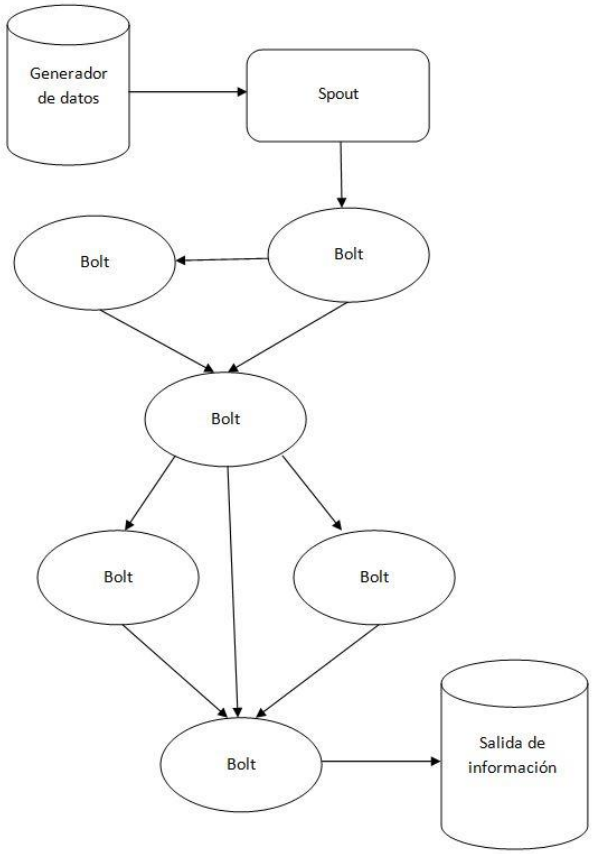


Ilustración 1: Diagrama Storm

A continuación, pasaremos a describir los diferentes elementos que componen una topología de Apache Storm:

- **Topology:** La lógica de una aplicación *Storm* en tiempo real, está constituida en una topología. Conjunto de *Spout* y *Bolts.*

- **Tupla:** Es la estructura de datos principal de *Storm*. Consiste en una lista ordenada de elementos de cualquier tipo, generalmente son valores separados por coma y traspasados a un *clúster* de *Storm*.

- **Stream:** Es una secuencia desordenada de *tuplas* que es procesada y creada en paralela de forma distribuida. Por defecto las *tuplas* pueden contener *integers, longs, shorts, bytes, strings, doubles, floats, boolean* y *arrays* de *bytes*. También se pueden definir otros tipos de datos de manera serializada para que puedan ser usados de forma nativa.

- **Spout:** Se encarga de leer las *tuplas* de una fuente externa y emitirlas dentro de la topología. Los *spouts* pueden ser *seguros* o *inseguros*. Un *spout seguro* es capaz de repetir una tupla si Storm falla al procesarla. Sin embargo esto no lo hace un *spout inseguro*, un *spout* de este tipo se olvida de la tupla que no ha podido ser procesada y pasa a la siguiente. Los *spouts* pueden emitir más de un *stream* de datos.

- **Bolt:** Los *bolts* son unidades de procesamiento lógicas. El *spout* envía los datos a los *bolts*, estos procesan las tuplas obtenidas y producen un nuevo *stream* de salida que emite las tuplas ya procesadas. El *stream* puede ser dirigido a uno o a varios *bolts*. Los *bolts* pueden realizar multitud de operaciones con los datos obtenidos del *stream* como son filtrar, agregar, unir, interactuar con bases de datos…

- **Task:** Se le llama *task* a la ejecución de una instancia de un *spout* o un *bolt*. Un *spout* o un *bolt* puede tener múltiples instancias ejecutándose en diferentes hilos de ejecución.

- **Workers:** Son los hilos de ejecución que puede tener una topología.

- **Stream Grouping:** Define como las tuplas de un *stream* deben ser particionadas entre los *bolts* de la topología. Hay 8 formas de agrupar las tuplas:

    - **Shuffle Grouping:** Las tuplas se distribuyen de forma aleatoria entre los *bolts* de forma que cada *bolt* recibirá la misma carga de trabajo.

    - **Fields Grouping:** Las *tuplas* se particionan por campos especificados. Las *tuplas* que tengan los mismos valores en los campos especificados (ej. id-número) serán asignados al mismo *bolt*.

    - **Partial Key Grouping:** Como la *Fields Grouping*, pero equilibrando la carga entre dos *bolts*. Esto proporciona un mejor aprovechamiento de los recursos, ya que si todos os valores son iguales, solo trabajaría un *bolt*. De esta forma se reparte la carga de trabajo.

    - **All Grouping:** Las *tuplas* son replicadas en todos los *bolts*.

    - **Global Grouping:** Todas las *tuplas* van a un solo *bolt*. Concretamente el que tenga el id más bajo.

    - **None Grouping:** No se especifica ninguna forma de partición. Storm aplica el *Shuffle Grouping* y si hay algún *bolt* sin carga de trabajo, se deshabilita.

- **Direct Grouping:** Es un tipo especial de agrupamiento. Especifica que el productor de las *tuplas* (el *bolt* anterior o el *spout)* decide que *bolt* será el encargado de procesarlas. Solo puede ser declarado en *Direct Streams.*

- **Local o Shuffle Grouping:** Se comporta igual que *Shuffle Grouping*, con la diferencia que solo se reparte la carga de trabajo entre los *bolts* que están activos. Si todos los *bolts* están activos, actúa igual que *Shuffle Grouping.*

Todos estos componentes, forman parte de una topología de *Storm*. Para poner en marcha una topología, primero se deben obtener datos en tiempo real mediante un *stream* (en nuestro caso Twitter).

El *stream* de datos debe estar conectado al primer elemento de la topología, el *spout.* A partir de este punto, la topología ya dispone de datos que están listos para ser procesados.

Para procesar los datos, dependiendo de la carga de trabajo que tenga el sistema, el número de tareas a realizar, o las diferentes etapas por las que tengan que pasar las tuplas, se crearán *bolts* en menor o mayor cantidad a los que el *spout* enviara los datos mediante un *stream* para que luego el *bolt* que recibe el *stream* se lo envíe a otro *bolt* y así sucesivamente hasta obtener los datos deseados.

Una vez creado los *bolts*, a cada *bolt* o conjunto de ellos, se le asignará un proceso a ejecutar, como por ejemplo puede ser almacenar todos los datos que vayan llegando en una lista que posteriormente será mostrada por pantalla.

Cuando las tuplas hayan recorrido toda la topología, pasando por todos los *bolts*, cada uno con una tarea asignada, se obtendrá la información procesada lista para ser usada, por ejemplo, almacenándola en base de datos o mostrándola por pantalla al usuario.

2.1.5. Arquitectura del Clúster

Una de las principales características de *Apache Storm* es que puede ser un sistema distribuido. Se puede instalar en tantos equipos como sea necesario para aumentar la capacidad y los recursos de la aplicación.

Un sistema distribuido, está formado por clústeres. Un *clúster es una* red de equipos conectados entre sí que funcionan como uno solo.

Un *clúster* de Apache Storm, está diseñado tal y como puede apreciarse en la *ilustración 2*:

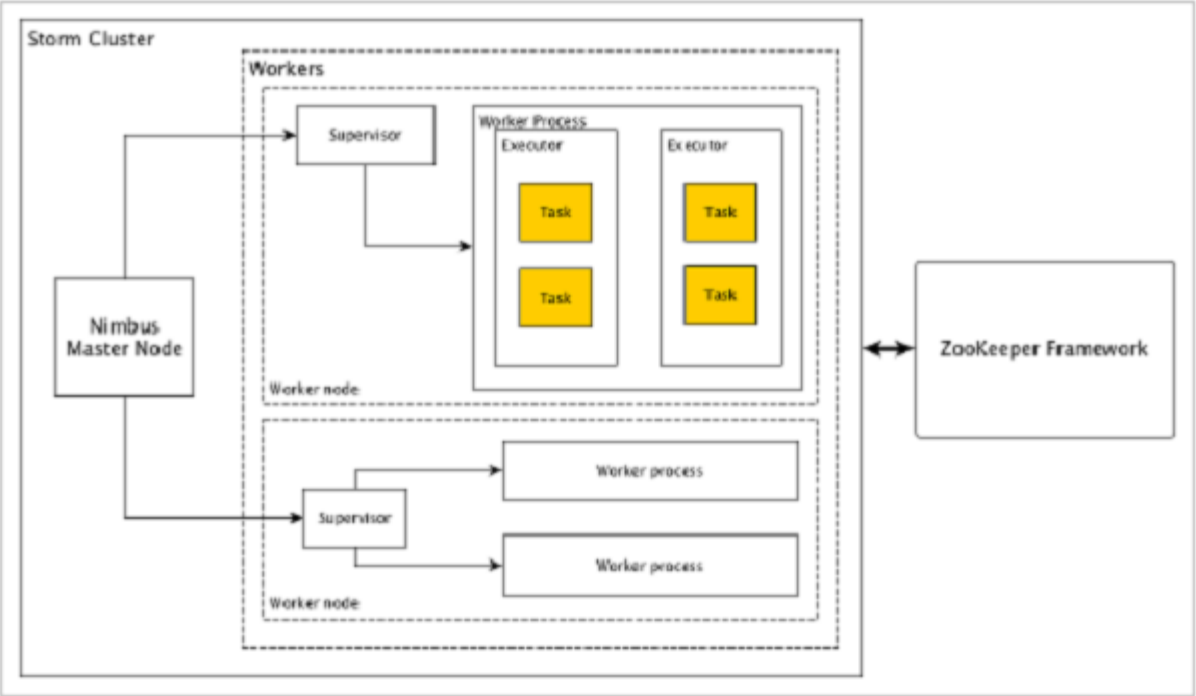


Ilustración 2: Arquitectura clúster Storm

Apache Storm tiene dos tipos de nodos, **Nimbus**, que es el *master node*, y **Supervisor**, que es el *worker node.* A continuación se describen los distintos elementos que conforman el *clúster*:

- **Nimbus:** Es el *master node* y el componente principal del *clúster* de *Storm*. Su función principal es ejecutar la *topología* de la aplicación. *Nimbus* analiza la *topología* y obtiene las tareas que tienen que ser ejecutadas. Luego distribuye estas tareas entre los nodos *Supervisor*.

- **Supervisor:** Obtiene las tareas de *nimbus.* Tiene varios *worker process* y se encarga de asignarles las tareas obtenidas de *nimbus* y que sean completadas satisfactoriamente.

- **Worker Process:** Es el encargado de ejecutar las tareas asignadas por el *supervisor*, pero no las ejecutará por sí solo, sino que creará *executors* y les pedirá que realice.

- **Executor:** Un *executor* es un hilo de ejecución que es generado por un *worker process.* Es capaz de ejecutar una o más tareas (*task)* pero siempre de un *bolt* o *spout* específico.

- **Task:** Realiza el procesamiento de datos.

- **Framework Zookeeper:** *Zookeeper* es un framework utilizado por el *clúster* para coordinar los nodos del mismo y mantener una sincronización de datos segura. Se encarga de mantener la comunicación entre *nimbus* y *supervisor.* Esto permite que

se pueda restablecer la conexión y continuar la ejecución en el punto en el que se paró cuando *nimbus* falle.

En conclusión, los elementos importantes que tiene un *clúster* de *Apache Storm* son un *nimbus, supervisors* y *Apache Zookeeper* que se encarga de establecer la coordinación entre *nimbus* y *supervisor*.

Para establecer un sistema distribuido con estos elementos, habría que distribuirlos en entre las maquinas que forman el *clúster*, tal y como se ve en la *ilustración* 3:

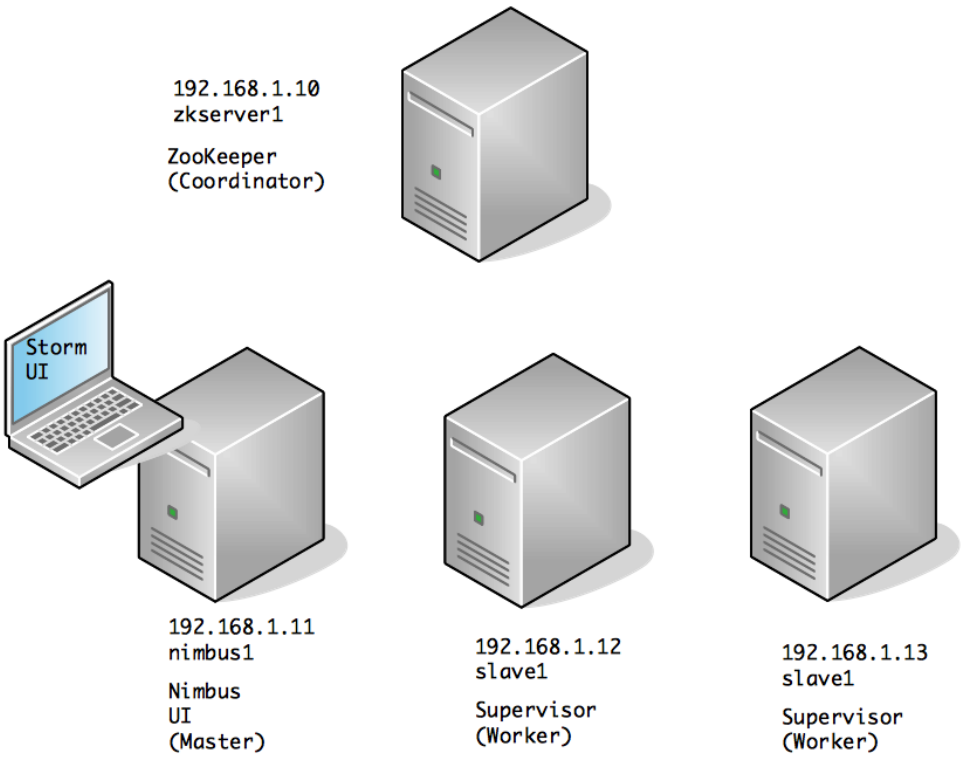


Ilustración 3: Distribución de elementos en clúster

A continuación veremos cómo es el funcionamiento del *clúster* paso a paso:

- Primero, *nimbus* esperara que la topología sea registrada en el *clúster*.
- Una vez registrada la topología, *nimbus* la procesa y obtiene todas la tareas que se van a realizar y el orden en el que se ejecutará cada una.
- Seguidamente, *nimbus* distribuye uniformemente las tareas entre todos los *supervisors* disponibles.

- Mientras que los *supervisors* están en ejecución envían señales a *nimbus* para indicarle que están "vivos".

- Cuando un *supervisor* deja de enviar señales, significa que está "muerto", entonces *nimbus* asigna las tareas restantes a otro *supervisor*.

- Si *nimbus* "muere" los *supervisors* siguen su ejecución sin ningún problema. *nimbus* será reiniciado por los servicios de monitorización.

- Una vez realizada todas las tareas que se le han asignado, el *supervisor* espera a que se le asigne una tarea nueva.

- *Nimbus,* una vez reiniciado, continúa su ejecución justamente desde el punto en que se quedó antes de pararse. De igual modo, el *supervisor* también puede reiniciarse una vez caído y continuará su ejecución por donde se había parado. Esto garantiza que todas las tareas en *Storm* sean ejecutadas al menos una vez.

Una vez se hayan ejecutado todas las tareas de la topología, *nimbus* espera a que se registre una nueva topología y el *supervisor* espera a que se le asigne una nueva tarea.

2.2. Red social

Las redes sociales son aplicaciones web en las que se reúnen comunidades de personas con actividades, gustos o intereses en común y que permiten que estas personas estén en "contacto virtual" de forma que puedan publicar, compartir, recibir o intercambiar información.

2.2.1. Tipos de redes sociales

No hay unos tipos especificados ni una clasificación para las redes sociales, pero viendo las características de las más principales, podríamos clasificarlas en **genéricas**, **profesionales** o **temáticas**.

Las redes sociales **genéricas** son las más numerosas y populares. En ellas no hay personas con gustos específicos y ni un tema principal del cual se tiene que tener un mínimo conocimiento. Estas redes sociales son usadas para conocer gente nueva, chatear con amigos o no tan amigos, colgar fotos a modo de álbum, o simplemente compartir información de noticias, videos, o álbumes de fotos de otros usuarios. En esta categoría se pueden incluir redes como **Facebook** o **Twitter.**

Otra categoría puede ser las redes sociales **profesionales**. Estas redes sociales son principalmente para introducirte en el mercado laboral. Aquí los usuarios suelen rellenar formularios con sus principales habilidades o aptitudes, además de compartir su currículo con el fin de que otros usuarios que pertenezcan a alguna empresa o trabajo los vean y

contacte con ellos. En esta categoría podrían estar redes como **LinkedIn** o **InfoJobs**, entre otras.

La última categoría sería las redes sociales **temáticas.** A estas redes sociales pertenecen usuarios que tienen los mismos gustos como pueden ser un tipo de deporte, un tipo de música, la pintura, o la fotografía. Aquí se comparte solo información que esté relacionada con la temática de la red social a la que se pertenezca. A esta categoría podría pertenecer la red social **Flickr**, cuya temática es la fotografía.

2.2.2. Algunas estadísticas

Para saber la magnitud de los datos que giran alrededor de las redes sociales, a continuación se muestran algunos datos estadísticos pertenecientes al año 2016.

Para establecer la relación, es interesante saber que la **población mundial** en marzo de 2016 era alrededor de **7,4 mil millones** de personas.

Internet tiene **3,17 mil millones** de usuarios activos de los cuales **2,3 mil millones** de usuarios son activos en redes sociales. Cada usuario tiene de promedio **5,54 cuentas** en diferentes redes sociales. Se ha registrado **176 millones** de usuarios nuevos en el año 2016 con respecto al año anterior. Hay **1 millón** de usuarios nuevos desde plataformas móviles cada día, es decir, **12 cada segundo**. Las principales plataformas de mensajería instantánea intercambian más de **60 mil millones** de mensajes diarios.

Además, como consecuencia de estos datos, las empresas aprovechan para invertir en publicidad puesto que los anuncios llegan a toda esa cantidad de personas cada día. Lo que genera que las redes sociales obtuvieron unos ingresos por publicidad de **8,3 millones de dólares** en 2015 [9].

2.3. Twitter

Una de las redes sociales más utilizadas por los usuarios de todo el mundo es Twitter. Twitter es, básicamente, una red social que consiste en escribir mensajes de, como máximo, 140 caracteres en los que se pueden incluir fotos, videos o enlaces a otros sitios de la web.

Sin embargo, comparado con otras redes sociales, las relaciones entre los usuarios son distintas, se podría decir que no son simétricas. La relación entre usuarios se diferencia entre **seguidores** (*followers)* y **seguidos** (*followed*).

Cuando alguien decide “seguirte” a ese usuario le llegarán todos los mensajes (*tuits*) que tú escribas y los verá en su cronología (*timeline*). El *timeline* es una sucesión de *tuits* ordenados cronológicamente en el que se muestran solo los *tuits* que han escrito los usuarios a los que se sigue. Es decir, en el *timeline* de cada usuario solo se verán los *tuits* que ese usuario quiere ver, porque ha decidido “seguir” a la persona que los escribe. Esto provoca que cada *timeline* sea distinto y que lo que lee el usuario cada día sea una información completamente personalizada acorde con sus gustos.

Además, en Twitter se ha creado un término llamado **"Trending Topic"** que en castellano viene a significar **tendencia** o **tema del momento**. Twitter te muestra una lista de tendencias del momento, esto significa que te muestra cual es el tema principal del que se está hablando en el mundo, en tu país o el área geográfica que selecciones, en este momento. Esto permite que estés al día de la actualidad a todas horas [3].

2.3.1. Algunas estadísticas

Para poder saber el impacto que tiene Twitter en el mundo, a continuación se muestran una serie de estadísticas que se han sido recogidas en el año 2016 [1].

Existen **310 millones** de usuario activos en Twitter cada mes. En total, se han creado **1,3 mil millones** de cuentas diferentes. En Estados Unidos, un **29,2%** de los usuarios de las redes sociales, son usuarios de Twitter. El **80%** de los usuarios activos, acceden a través de una plataforma móvil. El promedio de seguidores de un usuario de Twitter es **208 seguidores**. El usuario que más seguidores tiene es **Katy Perry** (cantante estadounidense) con **87 millones de seguidores**. El **83%** de los **líderes mundiales** tienen cuenta en Twitter.

Se estima que se mandan **500 millones** de *tuits* al **día**. Por lo que se mandan unos **6000 *tuits* por segundo.** Cada día, se produce suficiente contenido en Twitter como para llenar un libro de **10 millones de páginas**.

El **58%** de las **marcas publicitarias** tienen más de **100000 seguidores**. El **92% de las empresas** tuitean **1 vez al día**. Un usuario promedio de Twitter sigue a **5 empresas.** Las empresas que usan Twitter para dar **servicio al cliente** incrementa la satisfacción hasta un **19%.**

3. Planificación

En este capítulo se definirán las fases en las que se ha llevado a cabo el desarrollo del proyecto además de un diagrama de *Gantt* que ilustrara de una forma más gráfica cómo han sucedido estas fases a lo largo del tiempo.

3.1. Metodología de desarrollo

Debido a que la tecnología principal usada para el desarrollo del proyecto es reciente, no se ha especificado en qué consistirían las funcionalidades que es capaz de realizar el proyecto hasta que no se ha realizado un trabajo de investigación y se han ido descubriendo todas las características y posibilidades que podía ofrecer *Apache Storm*, por lo que el **modelo de ciclo de vida** que más se ajusta al proceso de desarrollo de la aplicación sería el modelo de ciclo de vida **incremental**.

El modelo de ciclo de vida incremental consiste, básicamente en ir añadiendo nuevas funcionalidades a las versiones del proyecto funcionales. Es decir, el proceso de desarrollo del proyecto se ha basado en ir terminando etapas de desarrollo e ir añadiendo nuevas funcionalidades a partir de las que ya existían en el proyecto.

3.2. Planificación del proyecto

A continuación se detallarán en que ha consistido cada fase que ha formado parte del proceso completo del desarrollo del proyecto y el trabajo que se ha realizado en cada una de ellas.

3.2.1. Primera fase: Elección de la tecnología

El proyecto comienza con una reunión con los tutores. En esta reunión se comunica la intención de realizar un proyecto que esté relacionado con el tema de *Big Data*, *Internet of Things* (Internet de las Cosas) o procesamiento de datos en tiempo real.

Surge la idea de usar una tecnología relativamente novedosa, teniendo en cuenta la velocidad con la que evolucionan las tecnologías en los tiempos que corren, una tecnología llamada *Apache Storm*.

Según la información que se sabía sobre esta tecnología, *Apache Storm* era un sistema de procesamiento de datos capaz de procesar información en tiempo real y la novedad era que la cantidad de información que podía procesar era enorme comparado con lo que se había trabajado hasta ahora en el departamento.

El objetivo al concluir esta primera reunión era realizar un trabajo de investigación para conocer más acerca de esta tecnología.

3.2.2. Segunda fase: Trabajo de investigación sobre Storm

Tras la primera reunión, se llevó a cabo una investigación acerca del uso de la tecnología a usar en el proyecto, es decir, aprender en qué consistía realmente, como se utiliza y para que se podía utilizar.

El trabajo de investigación fue duro y tomó un tiempo mayor al previsto debido a que la tecnología había surgido hacía relativamente poco tiempo y disponía de documentación oficial, pero no se disponía apenas de ningún ejemplo de uso o la posibilidad de ver el sistema funcionando en ningún escenario ya creado.

Con el paso de los días se fue conociendo más acerca de *Storm* y se pudieron ir realizando algunos ejemplos de funcionamiento con el fin de conocer mejor sus posibilidades y poder encontrar una temática adecuada con la que poder hacer funcionar correctamente el sistema aprovechando todas sus características.

3.2.3. Tercera fase: Elección de la temática

Tras conocer un poco más sobre las posibilidades de la tecnología que se iba a usar para la realización del proyecto, se concertó otra reunión con los tutores para comunicar lo que se había aprendido acerca de *Storm* y obtener una conclusión acerca de la temática a elegir para poder aplicar la tecnología.

El principal objetivo era elegir un tema en el cual se produjese información, la información sería procesada en el momento en el que se producía y se obtendría un resultado, como por ejemplo, una notificación.

Primero se pensó en obtener información acerca de la calidad del aire de diferentes regiones geográficas y obtener una notificación en cuanto se produzca una anomalía en los datos obtenidos.

También se pensó en hacer el mismo proceso que con la calidad del aire pero, en vez de esos datos, obtener los datos de la temperatura ambiental de distintas localizaciones e igualmente producir una notificación en el caso de que se detecte alguna anomalía en los datos obtenidos.

Tras debatir el tema, se llegó a la conclusión de que con estas dos temáticas no se aprovecharía el potencial que realmente tenía *Storm* puesto que los datos producidos en ambos casos, se producen cada cierto intervalo de tiempo y la cantidad de información producida es escasa en comparación con las posibilidades de procesamiento de las que presume ésta tecnología.

Finalmente se eligió como temática Twitter. Twitter es capaz de producir grandes cantidades de información en tiempo real, como son los *tuits* escritos por los usuarios de todo el mundo. Pero, ¿de qué se puede informar obteniendo los datos de Twitter?

Era necesario realizar otro trabajo de investigación acerca de los datos de la red social que son accesibles y de qué forma se pueden usar.

3.2.4. Cuarta fase: Trabajo de investigación sobre Twitter

Tras la elección de la temática de la que se van a obtener los datos que van a ser procesados con *Storm*, había que saber cómo obtener esos datos y qué datos obtener para poder realizar un procesamiento con una consecuente notificación.

Durante el trabajo de investigación, se descubre que existe una librería no oficial para el lenguaje de programación *Java* que permitía usar el API de Twitter llamada **twitter4j** [8].

Obteniendo esta librería ya seríamos capaces de conectar Twitter con *Storm* y acceder a la información que la red social nos produciría en tiempo real.

3.2.5. Quinta fase: Conexión de Storm con Twitter

En esta fase del desarrollo del proyecto, tras haber indagado en el API de Twitter, se descubre que Twitter proporciona un *stream* de datos para obtener los *tuits* que se van produciendo en tiempo real.

Conectando el *stream* de Twitter al *spout* de *Storm* se pudo realizar la conexión y por tanto se había conseguido tener acceso a los datos producidos por los usuarios de Twitter.

Se realizó una pequeña aplicación en la que se obtenía todos los *tuits* que se iban produciendo en formato *Json*. Analizando estos datos, se podía ver la información que se podía obtener de cada *tuit* haciendo uso de la librería antes mencionada.

3.2.6. Sexta fase: Objetivo del proyecto

Una vez vista toda la información que se puede obtener de un *tuit*, se decide cual va a ser el objetivo definitivo del proyecto.

El proyecto consistirá en una aplicación web, que permitirá a un usuario introducir unos parámetros de búsqueda, con estos parámetros se realizará una filtración y se obtendrán solo los *tuits* que cumplan con los requisitos introducidos. Una vez se tengan los *tuits* se accederán a su geolocalización, es decir, desde donde se ha escrito ese mensaje, y se mostrará en un mapa, coloreando la región geográfica a la que pertenezca.

3.2.7. Séptima fase: Procesado de tuits

En primer lugar se obtienen, mediante un formulario, los parámetros que se desean establecer en la búsqueda de los *tuits* y usando una función del API de Twitter, realizaremos el filtrado para que el *stream* nos proporcione los *tuits* que cumplan con esas especificaciones.

Una vez obtenidos, se extrae la información que necesitamos, en este caso sería el usuario que ha escrito el *tuit*, el mensaje que contiene, y la localización.

La localización se puede obtener de dos formas, mediante el nombre del lugar en el que se encuentra o al que pertenece el usuario, o mediante geolocalización, que nos ofrece las coordenadas exactas del lugar del que procede el mensaje.

En este punto del proyecto, se empieza generar la documentación correspondiente además de ir tomando notas del proceso y las herramientas utilizadas conforme irá avanzando el desarrollo del proyecto.

3.2.8. Octava fase: Obtención de la localización

Para obtener la localización del *tuit*, se toma la información que se ha adquirido de procesar el *tuit* que nos ha proporcionado el *stream* de Twitter. Esto se traduce en que tendremos una palabra o unas coordenadas que especifican la ubicación.

Para obtener la ubicación exacta, cotejaremos esos datos con la base de datos de *OpenStreetMap* [10], una aplicación parecida a Google Maps pero de software libre la cual nos proporcionará un Json el cual tendremos que procesar para obtener la información deseada.

Además, se tiene la posibilidad en el formulario, de establecer el nivel de detalle a nivel geográfico con el que se quiere mostrar la ubicación (por comunidades o por provincias de España) por lo que el tratamiento del Json será distinto para los distintos niveles seleccionados.

3.2.9. Novena fase: Almacenamiento de tuits

Una vez acabado el proceso de filtrado de *tuits* y teniendo los datos procesados, tendremos el usuario, mensaje y ubicación de cada *tuit* que haya cumplido con los parámetros de búsqueda que ha especificado el usuario.

Con este proceso, solo se podrían mostrar los resultados pero no se podría realizar ninguna otra tarea con los datos.

Para solucionar esto, almacenaremos los *tuits* en una lista conforme se vayan produciendo. El criterio de almacenado será agrupar los *tuits* en relación a su ubicación de forma que, se almacenarán todas las ubicaciones distintas que se hayan obtenido con el número de *tuits* que se han producido en cada una de ellas.

Esto nos permitiría obtener más estadísticas acerca de los datos obtenidos como son el número de *tuits* totales que se han registrado, el número de *tuits* que contiene cada región o el número de regiones desde las que se ha escrito un *tuit*.

3.2.10. Décima fase: Representación gráfica

Para hacer una representación gráfica del resultado de la búsqueda, se crea un mapa de España, utilizando para ello un formato de HTML llamado SVG. Con esto, lo que conseguimos es pintar una serie de una especie de "botones" que son *clickables* y tienen la forma y la posición de las distintas regiones del país.

Se realiza la vista de dos mapas, uno dividido en comunidades autónomas, y el otro dividido en provincias, que son las dos opciones que puede elegir el usuario en el formulario de búsqueda para representar los *tuits* obtenidos como resultado.

La función del mapa sería que cuando se produzca un *tuit*, y se obtenga los 3 datos antes especificados (usuario, mensaje y ubicación), se colocará la región del mapa que coincide con la ubicación del *tuit*. Los colores de las regiones serán distintos en función al número de *tuits* que se hayan registrado en ésta.

Como se ha descrito anteriormente, las regiones son *clickables*. Al pinchar sobre una región, se mostrarán los datos restantes (usuario y mensaje) de todos los *tuits* que se han registrado con esa ubicación.

3.2.11. Undécima fase: Actualizar mapa en tiempo real

Una vez realizada la representación gráfica, el resultado no fue del todo satisfactorio, ya que el usuario iniciaba la búsqueda, pero hasta que ésta no concluyera no se mostraba el mapa con las diferentes localizaciones. Esto se traduce en que, durante el tiempo que tardaba la aplicación en realizar el filtrado y el procesamiento de los datos, se mostraba una pantalla en la que solo se leía "Cargando…", lo que hacia la aplicación un poco lenta y poco interactiva.

Como solución, se habló con los tutores que la mejor idea era ir mostrando el mapa con los resultados conforme se iban obteniendo los diferentes *tuits*.

Esto se consiguió realizando el procedimiento de la topología de Storm en un hilo diferente de ejecución al hilo de ejecución que contiene los demás procesos de la aplicación como son el pintado del formulario de búsqueda o mapa.

Además, se añadió una función que consistía en recargar el mapa cada un intervalo de tiempo, por lo que siempre se mostraba el número de *tuits* y su ubicación de forma actualizada.

3.2.12. Duodécima fase: Parar la búsqueda

Otro de los inconvenientes de la aplicación era que una vez que el usuario especificaba el tiempo que se quería estar filtrando los *tuits* proporcionados por el s*tream* de Twitter, no se podía realizar otra acción hasta que se mostrase el resultado final de la búsqueda.

Esto se solucionó añadiendo un botón que interrumpiera el hilo de ejecución en el que se estaba procesando la topología de Storm. Esto provocaba que la búsqueda se diera por finalizada permitiendo así poder empezar otra búsqueda con otros parámetros completamente distintos.

3.2.13. Decimotercera fase: Sesiones y usuarios

En esta fase, se añade la posibilidad de crear sesiones individuales para cada usuario que utilice la aplicación ya que la aplicación está orientada para el uso en la web.

Para ello fue necesario hacer una reorganización del código de la aplicación al completo. Los principales cambios fueron el crear unas variables propias de cada sesión, con esto se consigue que no se comparta ninguna información entre los distintos usuarios que puedan estar usando la aplicación al mismo tiempo obteniendo así sesiones individuales.

3.2.14. Decimocuarta fase: Interacción con base de datos

En esta fase se añadirían varias funciones a la aplicación, lo que supondría tener interacción con base de datos.

Las funciones serían una pantalla de registro de usuarios, una pantalla de inicio de sesión y una pantalla de almacenamiento de búsquedas.

Como consecuencia y para poder establecer conexión con la base de datos, se ha implantado Hibernate en la aplicación. Hibernate, entre otras funciones, permite crear el esquema completo de la base de datos si al ejecutar la aplicación, ésta no existiese.

3.2.15. Decimoquinta fase: Validaciones de formularios

Una vez creadas las pantallas de inicio de sesión y de registro de usuario, se obtienen dos formularios con campo de texto en los que se puede escribir cualquier cosa.

Para obtener los datos correctos de los formularios se necesitan hacer validaciones de los datos introducidos por el usuario. Para hacer estas validaciones se ha utilizado JavaScript.

En la pantalla de registro de usuario, se tienen los campos de datos de usuario, los cuales hay que validar que no estén vacíos ya que son campos obligatorios; el campo de nombre de usuario, que tiene la validación de que el usuario tiene que ser único y por tanto no debe coincidir con un nombre de usuario que ya exista en la base de datos; y dos campos contraseña, contraseña y confirmación de contraseña, los cuales tiene que ser iguales en los dos campos.

De estos campos de registro de usuario, las validaciones de los campos se realizan en línea antes de enviar el formulario para así conseguir que la aplicación gane en rapidez. Una vez que los campos cumplen con todas la validaciones de estar todos rellenos y que coincidan los dos campos contraseña, se accede a la base de datos y se comprueba que el usuario no exista. Una vez completadas todas las validaciones, se registra un nuevo usuario en la base de datos.

Para la pantalla de inicio de sesión, se establecen dos campos de texto que serían usuario y contraseña. En primer lugar se comprueba que no haya ningún campo vacío con una validación en línea con JavaScript. Cuando los dos campos están llenos, se comprueba que el usuario introducido en el formulario existe en la base de datos, y en el caso de que existiera, se comprueba se la contraseña es correcta.

Si el usuario completa todas las validaciones, inicia sesión en la aplicación.

3.2.16. Decimosexta fase: Almacenar búsquedas

Se añade una función que consiste en que el usuario puede almacenar el resultado de una búsqueda una vez la haya realizado. Para esto, se ha incluido un botón guardar, que almacena los datos de la búsqueda correspondientes a fecha en la que se ha realizado la búsqueda, filtro (palabra o conjunto de palabras) que se ha utilizado para ejecutar la búsqueda, zoom o nivel de detalle geográfico con el que se quiere que se representen los *tuits* en el mapa (comunidades o provincias), tiempo que se ha establecido en los parámetros de la búsqueda, número de ubicaciones en los que se ha registrado un *tuit* y número de *tuits* totales que se han registrado.

Una vez almacenada la búsqueda, el usuario podrá acceder a su perfil, donde se mostrará toda su información personal, un formulario que permite cambiar la contraseña, el cual posee todas las validaciones anteriormente especificadas, y una tabla con todas las búsquedas que ese usuario ha almacenado.

3.2.17. Decimoséptima fase: Pruebas y correcciones

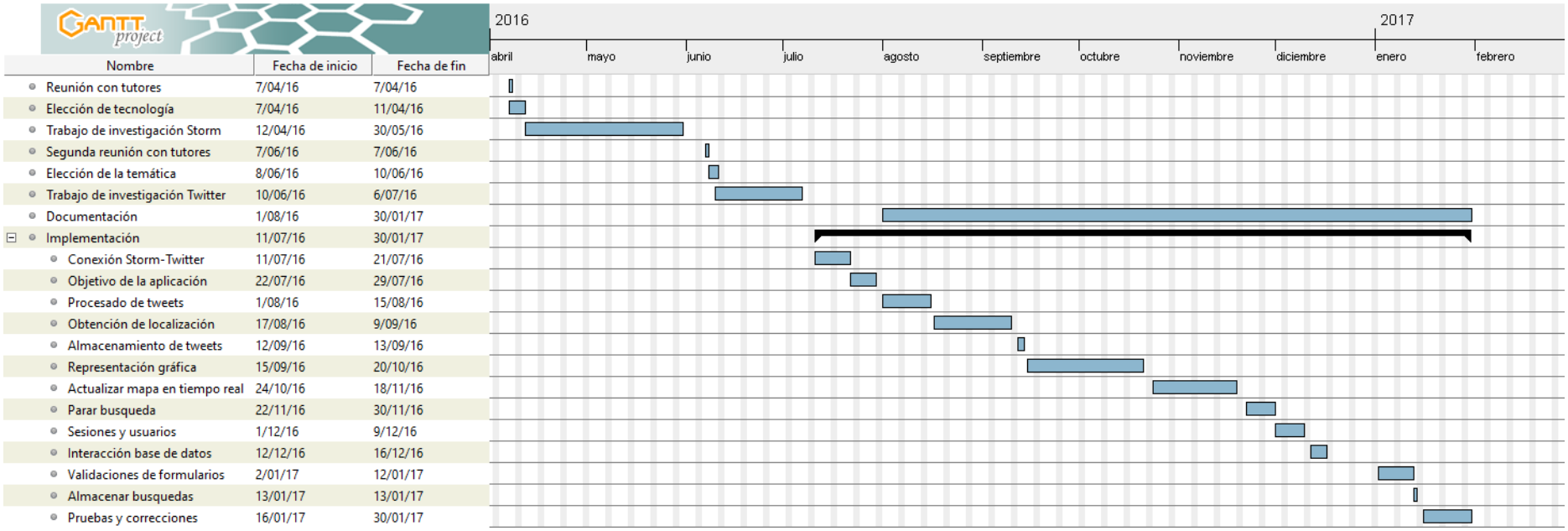
Como conclusión, y con la última versión de la aplicación estable y funcionando, se han realizado una serie de pruebas en cada una de las partes que componen el desarrollo completo de la aplicación.

Por último se han realizado las pertinentes pruebas de integración para comprobar que no existe ningún problema con la ejecución de la aplicación.

Se ha realizado las correcciones necesarias a los errores que se han detectado a través de la realización de dichas pruebas.

3.3. Diagrama de Gantt

A continuación se muestra un diagrama de Gantt realizado con la herramienta GanttProject [12] en el que se detalla el tiempo en el que se han realizado las diferentes fases del proyecto.



Nombre	Fecha de inicio	Fecha de fin
Reunión con tutores	7/04/16	7/04/16
Elección de tecnología	7/04/16	11/04/16
Trabajo de investigación Storm	12/04/16	30/05/16
Segunda reunión con tutores	7/06/16	7/06/16
Elección de la temática	8/06/16	10/06/16
Trabajo de investigación Twitter	10/06/16	6/07/16
Documentación	1/08/16	30/01/17
Implementación	11/07/16	30/01/17
Conexión Storm-Twitter	11/07/16	21/07/16
Objetivo de la aplicación	22/07/16	29/07/16
Procesado de tweets	1/08/16	15/08/16
Obtención de localización	17/08/16	9/09/16
Almacenamiento de tweets	12/09/16	13/09/16
Representación gráfica	15/09/16	20/10/16
Actualizar mapa en tiempo real	24/10/16	18/11/16
Parar busqueda	22/11/16	30/11/16
Sesiones y usuarios	1/12/16	9/12/16
Interacción base de datos	12/12/16	16/12/16
Validaciones de formularios	2/01/17	12/01/17
Almacenar busquedas	13/01/17	13/01/17
Pruebas y correcciones	16/01/17	30/01/17

Ilustración 4: Diagrama de Gantt

4. Descripción del proyecto

En este capítulo se realizará una descripción de las diferentes características y funcionalidades con las que cuenta este proyecto.

4.1. Funciones y características

Durante la realización de este proyecto, se ha tenido en cuenta al usuario a la hora de realizar la interfaz gráfica. Esto se traduce en que se ha realizado una interfaz bastante intuitiva y sencilla con el fin de que la aplicación pueda ser usada por todo tipo de usuario, ya sean personas más familiarizadas con el mundo de la tecnología o por el caso contrario no estén acostumbrados a usar este tipo de aplicaciones.

Las principales funciones del proyecto son:

- Detectar la escritura de un *tuit* en la red social Twitter y obtenerlo en tiempo real.
- Acceder a la información que nos proporciona un *tuit* como son el usuario que lo ha escrito, el texto que contiene o la ubicación desde donde se ha escrito.
- Filtrar la búsqueda de *tuits* mediante una consulta al *stream* proporcionado por Twitter.
- Realizar la representación gráfica de la ubicación de cada *tuit* en un mapa de España dividido en regiones.
- Establecer la palabra de filtrado del *stream*, el nivel de detalle geográfico con el que se quieren representar los *tuits* en el mapa y el tiempo que se desea que dure la búsqueda, mediante un formulario que rellenará el usuario.
- Mostrar el número total de *tuits* que se han obtenido al finalizar la búsqueda.
- Mostrar el número de ubicaciones desde las que se han escrito un *tuit* al finalizar la búsqueda.
- Mostrar el número de número de *tuits* que se ha escrito desde una región concreta al pinchar en esa región del mapa.
- Mostrar el contenido de todos los *tuits* que se hayan registrado en el mapa así como el nombre del usuario que lo ha escrito.
- Mostrar el proceso de rellenado de las distintas regiones del mapa en función a la ubicación de los *tuits* que se van obteniendo en tiempo real.

- Registrar usuarios y almacenarlos en la base de datos.
- Almacenar el resultado obtenido a partir de una búsqueda, realizada por un usuario, en la base de datos.
- Mostrar una tabla con todas las búsquedas que ha almacenado un usuario en la base de datos ordenadas por fecha.

4.2. Usuarios destino

En principio, la aplicación está desarrollada para cualquier tipo de usuario ya que el manejo de ésta es bastante sencillo e intuitivo.

Donde quizás se puedan distinguir tipos de usuario, es a la hora de usar la información que la aplicación proporciona a partir de las búsquedas realizadas y las búsquedas que almacena el usuario.

Seguramente, el uso más productivo que se le puede dar la información que produce la aplicación, sea con el objetivo de realizar algún tipo de estudio estadísticos como por ejemplo, si se quiere hacer un estudio sobre la elecciones del gobierno, se podría realizar una búsqueda sobre dónde y cuánto se habla de un partido político en España y así poder centrar las campañas en un lugar o en otro del país; o por ejemplo si se quiere abrir un negocio con que venda productos deportivos, se podría realizar una búsqueda de los distintos deportes y donde se habla más de ellos para poder buscar la mejor localización posible del negocio; o quizás para saber en qué lugar se está empezando a contagiar la gripe, entre otros posibles estudios que se podrían realizar.

4.3. Tecnologías usadas

A continuación, se muestran las diferentes tecnologías que se han usado para realizar el desarrollo del proyecto.

4.3.1. Apache Storm

Sistema de procesamiento de datos en tiempo real capaz de procesar enormes cantidades de información en un intervalo de tiempo muy corto.

Se ha usado para obtener los *tuits* producidos por los usuarios de Twitter en tiempo real y procesarlos hasta obtener los datos necesarios para poder realizar las diferentes funciones que es capaz de realizar la aplicación.

4.3.2. Twitter4j

Twitter4j es una librería no oficial de Java de código abierto que proporciona herramientas para manejar el API de Twitter [8].

El API de Twitter nos ofrece muchas posibilidades, entre ellas, nos permite obtener un s*tream* de datos que recoge toda la información producida en Twitter en tiempo real. Este *stream* será utilizado para conectarlo a nuestra topología Storm y poder procesar la información accediendo a ella mediante los métodos que nos proporciona este API.

Además del *stream*, el API nos permite acceder a los *tuits* que se van produciendo, y dentro de los *tuits*, nos permite acceder al usuario que lo ha escrito, a la localización desde el que se ha escrito y al mensaje que contiene, entre otros datos.

4.3.3. SVG

Gráficos Vectoriales Redimensionables (*Scalable Vector Graphics*) es un formato de gráficos vectoriales bidimensionales que te permite realizar formas tanto estáticas como animadas mediante la unión de líneas entre dos coordenadas especificadas en formato XML.

Esta herramienta se ha usado para presentar el mapa de España dividido en provincias y en comunidades autónomas.

Al crear los mapas con este formato, se crean divididos en regiones y además proporcionándole a estas regiones las propiedades de un botón de HTML. Esto quiere decir que las regiones del mapa son *clickables* y se les puede añadir un evento cuando se ejecuta el clic.

Igualmente se explicará más adelante de una forma más detallada el proceso de creación de los mapas.

4.3.4. Hibernate

Es software libre y es una herramienta de mapeo objeto-relacional que se utiliza para la plataforma Java que se encarga de realizar el mapeo entre el modelo de datos de una clase java y una base de datos relacional mediante archivos XML o mediante anotaciones en los atributos de la propia clase.

En este caso se ha usado para establecer el mapeo de los usuarios de la aplicación y las búsquedas que los usuarios tienen la posibilidad de almacenar en una base de datos MySQL.

4.3.5. Servidor Apache Tomcat

Apache Tomcat es un servidor que funciona como un contenedor de *servlets*. Es capaz de implementar las especificaciones de los *servlets* y de las *Java Server Pages* (JSP), ambos utilizados en el desarrollo de la aplicación web.

4.3.6. Base de datos MySQL

Base de datos encargada de almacenar los usuarios y las búsquedas que los usuarios deciden guardar. La conexión entre la aplicación y la base de datos se realiza mediante hibernate.

4.3.7. Lenguajes de programación

En esta sección se enumeran los lenguajes de programación usados para el desarrollo de la aplicación:

- **Java:** Es el lenguaje principal del proyecto. En este lenguaje se ha desarrollado toda la lógica de la aplicación.

    - **Timer:** Timer es una clase Java que permite lanzar una tarea creando un segundo hilo de ejecución al que se le puede asignar un tiempo de espera para empezar a ejecutarse y un número de repeticiones que son las veces que se ejecutará el código.

      En este proyecto se utiliza para realizar la búsqueda de *tuits* en segundo plano, mientras se va mostrando la representación gráfica del mapa en tiempo real.

    - **Servlet:** Es una clase de Java, utilizada para ampliar las capacidades de un servidor. Los *servlets* son utilizados para generar páginas web de forma dinámica a partir de los parámetros de la petición que envíe el navegador web.

    - **BufferedReader:** Se encarga de obtener datos de entrada, en este caso haciendo una consulta mediante una url a **OpenStreetMap**, la cual devuelve un Json con datos que será necesario procesar para obtener la ubicación exacta del *tuit*.

    - **JsonObject:** Se usa para procesar una serie de datos Json y obtener el campo deseado, en este caso la ubicación del *tuit*.

- **JavaScript** y **Ajax:** Se utiliza para realizar las transiciones entre las distintas pantallas de la aplicación y conseguir así que la navegación sea más dinámica para el usuario que la está usando.

- **HTML** y **CSS**: Se ha usado para realizar el maquetado de la interfaz gráfica de la aplicación y poder realizar una navegación sencilla e intuitiva para el usuario.

4.3.8. Software

Para desarrollar el proyecto, se ha utilizado software que se nombrara a continuación cuya función es agilizar el proceso de implementación del mismo.

- **Eclipse:** Es un entorno de desarrollo que agiliza el proceso de creación de código de programación para el desarrollo de la aplicación.

- **Maven:** Es una herramienta de software para la gestión y construcción de proyectos java. Utiliza un *Project Object Model* (POM) para describir el proyecto a construir, sus dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos. Se encarga de realizar la compilación del código y su empaquetado.

- **Git:** Es un software de control de versiones que permite realizar copias de seguridad del estado actual del proyecto en la nube. En este caso se han ido realizando copias de seguridad progresivas en un repositorio de la plataforma **Neptuno** que pertenece al grupo de investigación **UCASE**.

4.3.9. Hardware

El proyecto se ha desarrollado en su totalidad en un ordenador portátil Acer Aspire 5733 con un procesador Intel Core i3 y con una memoria RAM de 4GB, en el cual se ha realizado tanto el desarrollo como las pruebas.

La aplicación se despliega en un Servidor Local Apache Tomcat por lo que no corresponde a este apartado, al igual que la base de datos.

5. Requisitos del sistema

En este capítulo se realizará una descripción detallada de los requisitos y los objetivos que debe cumplir la aplicación una vez desarrollada.

5.1. Objetivos del sistema

El objetivo principal del sistema es realizar un filtrado de los *tuits* que se vayan produciendo en tiempo real y ser capaz de situarlos en un mapa, obteniendo la información de la ubicación que indica desde donde se ha realizado el *tuit*.

Además ser capaz de registrar usuarios y almacenar el resultado de las búsquedas que un usuario haya realizado cuando éste lo desee. La lista de búsquedas debe estar relacionada con cada usuario.

5.2. Catálogo de requisitos

A continuación se describen los diferentes requisitos de la aplicación.

5.2.1. Requisitos funcionales

Los requisitos funcionales que debe cumplir la aplicación son:

- **Registrar un usuario:** La aplicación debe ser capaz de registrar un usuario nuevo, almacenarlo en la base de datos y que el nombre de usuario de todos los usuarios registrados sean únicos.

- **Iniciar sesión:** El sistema debe ser capaz de iniciar una sesión para cada usuario obteniendo su nombre de usuario, comprobando que el usuario existe en base de datos, obteniendo su contraseña y comprobando que sea correcta.

- **Cerrar sesión:** El sistema debe ser capaz de eliminar la sesión de un usuario que ya haya accedió al sistema mediante usuario y contraseña.

- **Gestionar contraseña:** EL sistema debe ser capaz de ofrecer a un usuario, que haya iniciado sesión, cambiar su contraseña, introduciendo su contraseña actual, la nueva contraseña y la confirmación de la nueva contraseña.

- **Realizar búsquedas parametrizadas de *tuits:*** El sistema debe ser capaz de obtener una lista de *tuits* realizando una consulta parametrizada mediante un formulario que haya realizado el usuario anteriormente.

- **Situar los *tuits* en un mapa:** El sistema debe ser capaz de representar gráficamente la ubicación de los *tuits* en un mapa dividido en regiones mediante el acceso a los datos proporcionados por el propio *tuit* obtenido.

- **Interrumpir la búsqueda:** El sistema debe ser capaz de interrumpir una búsqueda que el usuario haya puesto en marcha con un tiempo de búsqueda determinado, sin que éste llegue a cumplirse completamente, mostrando correctamente el resultado obtenido hasta el momento de la interrupción.

- **Mostrar número de *tuits*:** El sistema debe ser capaz de mostrar el número total de *tuits* obtenidos tras realizar la búsqueda parametrizada.

- **Mostrar ubicaciones:** El sistema debe ser capaz de mostrar la lista y el número de ubicaciones total desde las que se haya obtenido un *tuit* al realizar la búsqueda parametrizada

- **Mostrar lista de *tuits* de cada ubicación:** El sistema debe ser capaz de desplegar la lista de *tuits* que se han registrado en una región concreta, ya sea comunidad o provincia, cuando el usuario pinche en ésta región mostrando el usuario que ha escrito el *tuit* y el mensaje que contiene.

- **Almacenar resultado de búsquedas:** El sistema debe ser capaz de ofrecer al usuario la posibilidad de almacenar el resultado de la búsqueda que ha realizado. Una vez elegida la opción de almacenar la búsqueda, el sistema debe ser capaz de guardar los datos finales de la búsqueda en una base de datos.

5.2.2. Requisitos no funcionales

Los requisitos funcionales que debe cumplir la aplicación son:

- **Tiempo de respuesta:** La aplicación debe dar una respuesta rápida al usuario de forma que la navegación entre las interfaces sea fluida.

- **Rendimiento:** La aplicación debe funcionar fluidamente incluso cuando la cantidad de trabajo sea grande.

- **Disponibilidad:** La aplicación, al ser una aplicación web, depende de la conexión a Internet de la que disponga el usuario. De todas formas, en condiciones normales, la aplicación debe estar disponible para los usuarios las 24 horas al día los 7 días de la semana.

- **Escalabilidad:** La aplicación debe soportar que en un futuro se le realizan ampliaciones tanto de máquinas que ayuden a realizar tareas como sistema distribuido como de funcionalidades.

- **Seguridad:** La aplicación debe mantener la seguridad y la integridad de los datos almacenados ya que contiene datos de usuarios.

- **Mantenibilidad:** El sistema debe ser fácilmente mantenible necesitando el menor número de modificaciones si el sistema fallara en algún momento. Este sistema, al estar desarrollado en *java* mediante una programación orientada a objetos, se obtiene un sistema con varias clases independientes con un nivel de acoplamiento bajo lo que conlleva a una mantenibilidad mucho más simple.

- **Fiabilidad:** El sistema debe ser completamente fiable y tolerante a fallos. Esto se consigue capturando las excepciones que se puedan producir y transformándolas en mensajes informativos. La aplicación debe quedar en un estado consistente.

5.2.3. Requisitos de información

Los requisitos de información que debe cumplir la aplicación son los siguientes:

- El sistema será capaz de almacenar usuarios en una base de datos relacional.

- El sistema será capaz de almacenar los datos de una búsqueda realizada en una base de datos, relacionando la búsqueda con un usuario existente.

- El sistema será capaz de mostrar toda la información de un usuario y ofrecerle la posibilidad de cambiar su contraseña.

- El sistema será capaz de evitar que al registrar un usuario, no se produzca ningún nombre de usuario duplicado.

5.2.4. Requisitos de interfaz

Los requisitos de interfaz que debe cumplir la aplicación son los siguientes:

- Un usuario que no haya iniciado sesión en la aplicación no podrá acceder a ninguna interfaz de la aplicación, que no sea la de *login*, de ninguna manera.

- Un usuario que no esté registrado no podrá iniciar sesión y por tanto no podrá acceder a ninguna interfaz de la aplicación.

- Un usuario que haya iniciado sesión puede navegar por todas las interfaces de la aplicación.

6. Análisis del sistema

En este capítulo se realizará un análisis detallado del sistema.

6.1. Modelo de Casos de uso

En esta sección se mostrarán las especificaciones de los diferentes casos de uso.

6.1.1. Casos de uso relacionados con el usuario

- **Caso de uso:** Registro.
    - **Descripción:** Registro de un nuevo usuario en la aplicación.
    - **Actores:** Usuario.
    - **Resumen:** Lo primero que debe hacer un usuario al abrir la aplicación es registrarse para obtener unas credenciales con las que loguearse.
    - **Precondiciones:** El usuario debe un usuario nuevo y disponer de conexión a Internet.
    - **Postcondiciones:** Se crea un usuario nuevo y el sistema mostrará un mensaje de confirmación de nuevo registro.
    - **Escenario principal:**
        - El usuario pincha en el botón “Registrarse”.
        - Aparece la pantalla de registro.
        - El usuario rellena los campos con la información correspondiente.
        - El usuario pincha en “Registrarse”.
        - El sistema valida que lo campos estén rellenos correctamente.
        - Se almacena los datos del usuario en la base de datos.
    - **Escenarios alternativos:**
        - El usuario puede cerrar la aplicación en cualquier momento.
        - El usuario deja algún campo vacío y se muestra un mensaje de error.
        - El nombre de usuario ya existe y se muestra un mensaje de que el usuario ya existe.
        - Los campos de contraseña y confirmar contraseña no son iguales.

- **Caso de uso:** Login.
    - **Descripción:** El usuario inicia sesión en la aplicación.
    - **Actores:** Usuario.
    - **Resumen:** El usuario inicia sesión introduciendo las credenciales, que previamente ha obtenido mediante el registro, para poder acceder a la aplicación.
    - **Precondiciones:** El usuario debe estar registrado en la aplicación con anterioridad y disponer de conexión a Internet.
    - **Postcondiciones:** El usuario inicia sesión y puede acceder al sistema.
    - **Escenario principal:**
        - El usuario introduce el usuario y contraseña y pulsa en "Iniciar Sesión".
        - El sistema comprueba que el usuario existe.
        - El sistema comprueba que la contraseña que corresponde al usuario es igual a la contraseña introducida.
        - Se crea una sesión con las credenciales del usuario y se le concede acceso.
    - **Escenarios alternativos:**
        - El usuario puede cerrar la aplicación en cualquier momento.
        - El nombre de usuario introducido no existe en la base de datos y se muestra un mensaje de error.
        - La contraseña introducida no es igual a la contraseña que corresponde con el nombre de usuario almacenado en la base de datos y se muestra un mensaje de error de contraseña

.

- **Caso de uso:** Buscar.
    - **Descripción:** El usuario inicia la búsqueda de *tuits*.
    - **Actores:** Usuario.
    - **Resumen:** El usuario rellena los parámetros necesarios para realizar la búsqueda de *tuits*.
    - **Precondiciones:** El usuario debe estar logueado y disponer de conexión a Internet. Al rellenar el formulario de parámetros, debe introducir un tiempo de búsqueda igual o mayor a 30 segundos.
    - **Postcondiciones:** Se inicia la búsqueda de *tuits* con los parámetros que ha introducido el usuario.

- **Escenario principal:**

  - El usuario con la sesión iniciada rellena los parámetros de búsqueda y pincha en "Buscar"
  - El sistema comienza el proceso de búsqueda de *tuits*.
  - El sistema termina el proceso de búsqueda y muestra los resultados.

- **Escenarios alternativos:**

  - El usuario puede cerrar la aplicación en cualquier momento.
  - La sesión del usuario ha expirado y tiene que loguearse de nuevo.
  - El usuario inserta un tiempo de búsqueda menor a 30 segundos y se muestra un mensaje de error de tiempo mínimo.
  - El usuario no rellena el campo "Filtro" y se buscan todos los *tuits* que se escriben desde España.
  - El usuario rellena el filtro introduciendo dos palabras separadas por un guion y el sistema realiza una búsqueda de los *tuits* la primera palabra o la segunda.
  - Durante la ejecución de la búsqueda el usuario pulsa el botón "Parar" y la búsqueda finaliza automáticamente y muestra los resultados.

- **Caso de uso:** Ver *tuits.*

  - **Descripción:** Ver los *tuits* de una ubicación.

  - **Actores:** Usuario, Sistema

  - **Resumen:** Ver la lista de *tuits*, su usuario y su mensaje, que pertenecen a la misma región.

  - **Precondiciones:** El usuario debe estar *logueado* y disponer de conexión a Internet. El usuario debe haber realizado una búsqueda y haber obtenido unos resultados.

  - **Postcondiciones:** Se despliega un panel con una lista de *tuits* que pertenecen a una misma región mostrando el usuario y el mensaje de cada *tuit.*

  - **Escenario principal:**

    - El usuario pincha sobre una región del mapa.
    - El sistema despliega un panel con el número de *tuits* de esa región y la lista de usuarios con el mensaje que contiene cada *tuit.*

  - **Escenarios alternativos:**

- El usuario puede cerrar la aplicación en cualquier momento.
- La sesión del usuario ha expirado y tiene que *loguearse* de nuevo.
- No hay ningún *tuit* registrado en esa región y se muestra un mensaje indicándolo.

- **Caso de uso:** Ver localizaciones.
  - **Descripción:** Ver la lista de localizaciones.
  - **Actores:** Usuario, Sistema
  - **Resumen:** Ver la lista de ubicaciones en las que se ha registrado un *tuit.*
  - **Precondiciones:** El usuario debe estar *logueado* y disponer de conexión a Internet. El usuario debe haber realizado una búsqueda y haber obtenido unos resultados.
  - **Postcondiciones:** Se despliega un panel con el número de localizaciones en las que se ha registrado un *tuit*, y una lista con el nombre de las localizaciones registradas y el número de *tuits* que se ha escrito en cada una de ellas.
  - **Escenario principal:**
    - El usuario pincha sobre el botón de "Tuits totales".
    - Se despliega un panel que indica el número de localizaciones, una lista con el nombre de las localizaciones y el número de *tuits* que se ha registrado en cada localización.
  - **Escenarios alternativos:**
    - El usuario puede cerrar la aplicación en cualquier momento.
    - La sesión del usuario ha expirado y tiene que *loguearse* de nuevo.
    - No se ha registrado ningún *tuit*, por lo que el número de localizaciones será cero y el panel estará vacío.

- **Caso de uso:** Guardar búsqueda.
  - **Descripción:** Almacenar búsqueda en base de datos.
  - **Actores:** Usuario.
  - **Resumen:** El usuario tiene la opción de almacenar los resultados obtenidos de la búsqueda que ha realizado en la base de datos.
  - **Precondiciones:** El usuario debe estar *logueado* y disponer de conexión a Internet. El usuario debe haber realizado una búsqueda y haber obtenido unos resultados.

- **Postcondiciones:** Se almacena en base de datos los resultados de la búsqueda realizada.

- **Escenario principal:**

  - El usuario ha realizado la búsqueda y el sistema ha devuelto los resultados.
  - El usuario pincha en el botón "Guardar".
  - El sistema almacena en la base de datos los resultados de la búsqueda obtenidos.
  - El sistema devuelve un mensaje de confirmación.

- **Escenarios alternativos:**

  - El usuario puede cerrar la aplicación en cualquier momento.
  - La sesión del usuario ha expirado y tiene que loguearse de nuevo.

- **Caso de uso:** Ver perfil.

  - **Descripción:** Ver perfil de usuario.

  - **Actores:** Usuario, Sistema

  - **Resumen:** El usuario pincha en su nombre y accede a una pantalla que le muestra sus datos de perfil.
  -
  - **Precondiciones:** El usuario debe estar logueado y disponer de conexión a Internet.

  - **Postcondiciones:** Se muestra una pantalla con los datos del usuario y la lista de resultados de las búsquedas que ha almacenado. Además se muestra un formulario de gestión de contraseña.

  - **Escenario principal:**

    - El usuario pincha en su nombre.
    - El sistema lo redirige a una pantalla en la que se muestran todos los datos del usuario así como los resultados de las búsquedas que ha almacenado.

  - **Escenarios alternativos:**

    - El usuario puede cerrar la aplicación en cualquier momento.
    - La sesión del usuario ha expirado y tiene que loguearse de nuevo.

- **Caso de uso:** Cambiar contraseña.

  - **Descripción:** El usuario modifica su contraseña.

- **Actores:** Usuario.
- **Resumen:** El usuario accede a la pantalla que muestra sus datos de perfil, y modifica la contraseña que utiliza para iniciar sesión.
- **Precondiciones:** El usuario debe estar logueado y disponer de conexión a Internet.
- **Postcondiciones:** El usuario cambia su contraseña de acceso y se muestra un mensaje de confirmación.
- **Escenario principal:**
  - El usuario pincha en su nombre.
  - El sistema lo redirige a una pantalla en la que se muestran todos los datos del usuario así como los resultados de las búsquedas que ha almacenado.
  - El usuario rellena los campos del formulario que le permite cambiar la contraseña.
  - El sistema modifica la contraseña en base de datos y muestra un mensaje de confirmación.
- **Escenarios alternativos:**
  - El usuario puede cerrar la aplicación en cualquier momento.
  - La sesión del usuario ha expirado y tiene que loguearse de nuevo.
  - El usuario no ha rellenado algún campo y se muestra un mensaje de error.
  - Los campos "Nueva contraseña" y "Confirmar nueva contraseña" rellenados por el usuario, con coinciden y se muestra un mensaje de error.
  - El usuario ha rellenado erróneamente el campo "Contraseña actual" y se muestra un mensaje de error.

- **Caso de uso:** Cerrar sesión.
  - **Descripción:** El usuario cierra sesión.
  - **Actores:** Usuario.
  - **Resumen:** El usuario pincha en "Cerrar sesión" y sale de la aplicación.
  -
  - **Precondiciones:** El usuario debe estar logueado y disponer de conexión a Internet.
  - **Postcondiciones:** Se cierra la sesión del usuario y se redirige a la pantalla de *login*.
  - **Escenario principal:**

- El usuario pincha en “Cerrar sesión”
- El sistema pone la sesión a null.
- Cuando se intenta acceder a alguna página, se le redirige a la pantalla de *login*.

- **Escenarios alternativos:**

  - El usuario puede cerrar la aplicación en cualquier momento.
  - La sesión del usuario ha expirado y tiene que *loguearse* de nuevo.

6.1.2. Casos de uso de relacionados con Storm

- **Caso de uso:** Conectar con Twitter.

  - **Descripción:** Conectar el *spout* de la topología de *Apache Storm* con el s*tream* de Twitter.

  - **Actores:** *Storm,* Twitter.

  - **Resumen:** El *spout* de *Storm* se conecta al *stream* de Twitter para obtener los datos que proporciona la plataforma.

  - **Precondiciones:** Disponer de conexión a Internet y obtener datos del *stream* de Twitter.

  - **Postcondiciones:** Se obtienen los datos de Twitter y se mandan los datos al siguiente *bolt.*

  - **Escenario principal:**

    - Twitter proporciona un *stream* de datos.
    - El *spout* de la topología de *storm* se conecta al *stream* de twitter.
    - Cada vez que se produce un *tuit*, éste se manda por el *stream* de datos.
    - El *spout* obtiene los datos del *tuit* mediante el *stream* y crea un *stream* dentro de la topología que se encargará de llevar los datos a los *bolts.*

  - **Escenarios alternativos:**

    - No se produce ningún *tuit* y por tanto no se obtendrá ningún tipo de información.

- **Caso de uso:** Obtener datos.

  - **Descripción:** Obtener los datos del *tuit.*

- **Actores:** *Storm*, Twitter.
- **Resumen:** Obtener los datos, de un *tuit*, necesarios para procesarlos en las siguientes fases.
- **Precondiciones:** Disponer de conexión a Internet y obtener datos del *spout*.
- **Postcondiciones:** Se obtienen los datos de ubicación, usuario y mensaje de cada *tuit* se mandan al siguiente *bolt.*
- **Escenario principal:**
  - El *bolt* obtiene todos los datos del *tuit* procedente del *spout*.
  - El *bolt* capta los datos necesarios como son: ubicación, usuario, y mensaje.
  - El *bolt* pasa los datos obtenidos al siguiente *bolt.*
- **Escenarios alternativos:**
  - No se produce ningún *tuit* y por tanto no se obtendrá ningún tipo de información.

- **Caso de uso:** Obtener ubicación.
  - **Descripción:** Obtener la ubicación del *tuit*.
  - **Actores:** *Storm, OpenStreetMap.*
  - **Resumen:** Se obtiene la ubicación exacta del *tuit* cotejando los datos obtenidos con la base de datos de *OpenStreetMap.*
  - **Precondiciones:** Disponer de conexión a Internet y obtener datos del *bolt*.
  - **Postcondiciones:** Se obtiene la ubicación exacta del *tuit* y se almacena el resultado en una lista.
  - **Escenario principal:**
    - El *bolt* obtiene los datos de ubicación del *tuit.*
    - El *bolt* manda una consulta a *OpenStreetMap.*
    - *OpenStreetMap* devuelve un *Json* con la información de la ubicación exacta.
    - El *bolt* procesa el *Json* y obtiene la ubicación exacta del *tuit.*
    - El *bolt* almacena la ubicación obtenida.
  - **Escenarios alternativos:**

        - No se produce ningún *tuit* y por tanto no se obtendrá ningún tipo de información.
        - La ubicación del *tuit* no existe y por tanto no se obtiene la ubicación.
        - Al almacenar la ubicación, se comprueba que ya se ha encontrado un *tuit* con esa ubicación, entonces se incrementa el contador individual de esa ubicación.

- **Caso de uso:** Obtención de mensaje y usuario.

    - **Descripción:** Obtención del usuario y mensaje del *tuit*.

    - **Actores:** *Storm.*

    - **Resumen:** Obtención y procesado del usuario que ha escrito el *tuit* y el mensaje que contiene.

    - **Precondiciones:** Disponer de conexión a Internet y obtener datos del *bolt*.

    - **Postcondiciones:** Se obtiene un mensaje formateado que contiene el usuario y el mensaje del *tuit y se almacena en una* lista.

    - **Escenario principal:**

        - El *bolt* obtiene el usuario y el mensaje del *tuit* que proviene del *bolt* anterior.
        - El *bolt* formatea el usuario y el mensaje de la forma en que se quiere mostrar por pantalla.
        - El *bolt* obtiene la lista de ubicaciones de todos los *tuits* que se han encontrado y almacena el mensaje en el lugar que le corresponde comparando la ubicación del *tuit* con la lista.

    - **Escenarios alternativos:**

        - No se produce ningún *tuit* y por tanto no se obtendrá ningún tipo de información.
        - No hay información acerca de la ubicación, por tanto el mensaje se almacenará en la lista de “Vacío”.

- **Caso de uso:** Representar tuit

    - **Descripción:** Representar un *tuit* gráficamente.

    - **Actores:** Sistema.

    - **Resumen:** Representar un *tuit* gráficamente mostrándolo en un mapa por pantalla coloreando la región a la que pertenece obteniendo su ubicación.

    - **Precondiciones:** Obtener una lista de localizaciones de España.

- **Postcondiciones:** Se obtiene un mapa de España con las regiones coloreadas de un color u otro dependiendo del número de *tuits* que se hayan registrado en cada una de ellas.

- **Escenario principal:**

  - El sistema obtiene la lista de ubicaciones registradas y el número de *tuits* de cada una de ellas.
  - El sistema colorea la región del mapa a la que corresponde la ubicación variando el color dependiendo del número de *tuits* que se obtengan.

- **Escenarios alternativos:**

  - No se produce ningún *tuit* y por tanto no se obtendrá ningún tipo de información.
  - La ubicación del *tuit* no corresponde a una ubicación de España y por tanto no se colorea ninguna región.

6.1.3. Diagramas de casos de uso

- Diagrama de casos de uso relacionados con el usuario.

Ilustración 5: Casos de uso Usuario

- Diagrama de casos de uso relacionado con *Storm*.

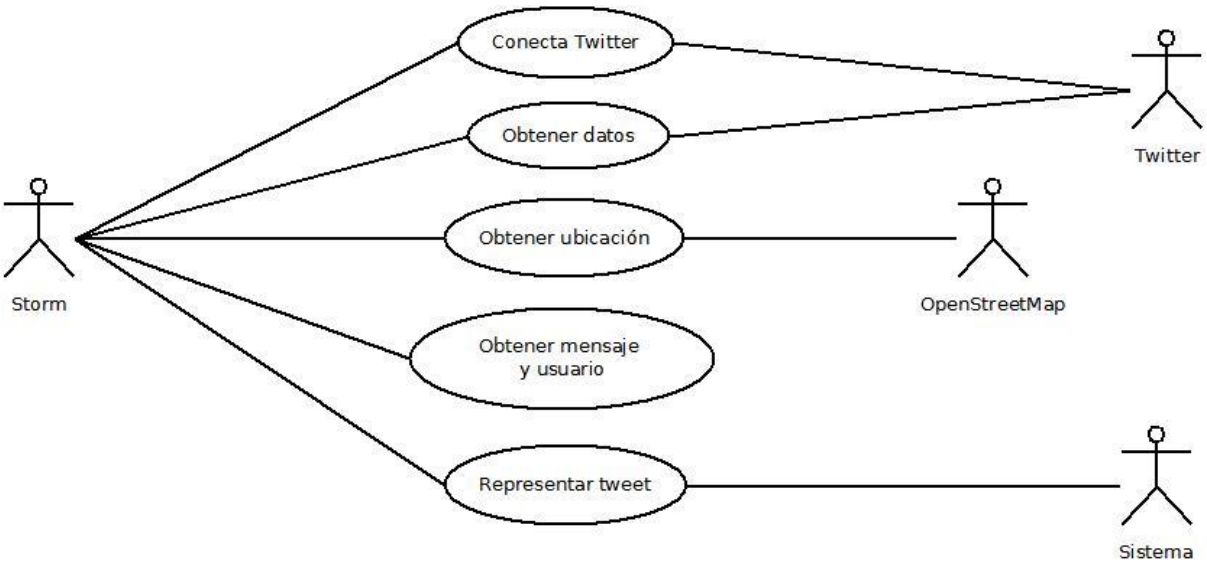


Ilustración 6: Casos de uso Storm

6.2. Modelo conceptual de datos

A continuación se muestra el diagrama del modelo conceptual de datos de la aplicación desarrollada.

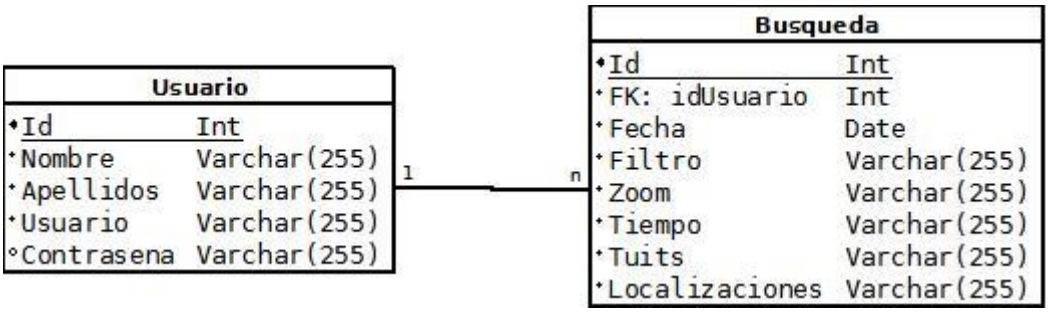


Ilustración 7: Modelo conceptual de datos

6.3. Diagramas de secuencia

A continuación se muestran los diagramas de secuencia de la aplicación desarrollada.

6.3.1. Diagramas de secuencia relacionados con el usuario

En esta sección se muestran los diagramas de secuencia de las operaciones en las que interfiere el usuario:

- Registro de un nuevo usuario

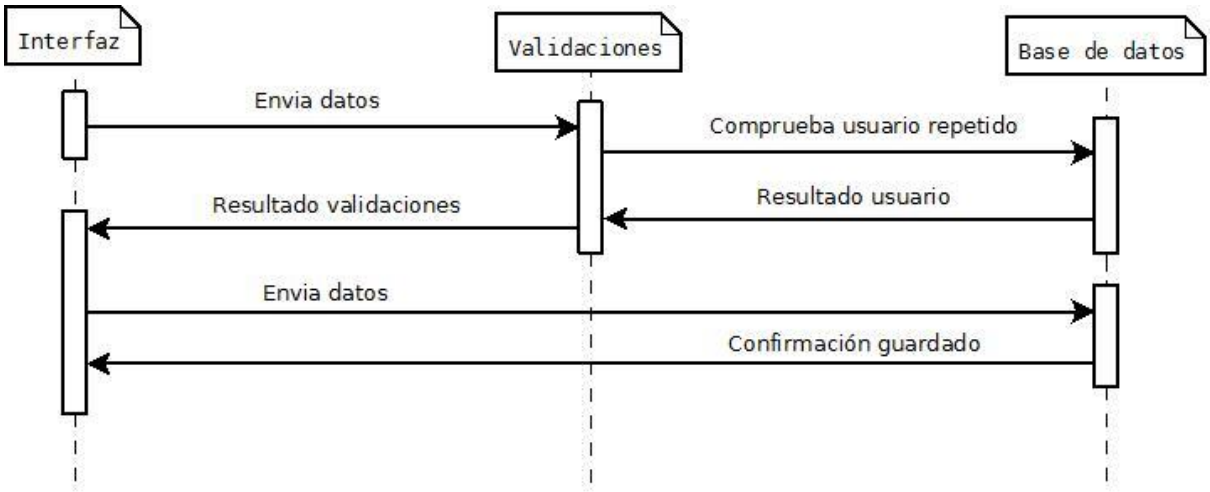


Ilustración 8: DS Registro de usuario

- Iniciar sesión en la aplicación.

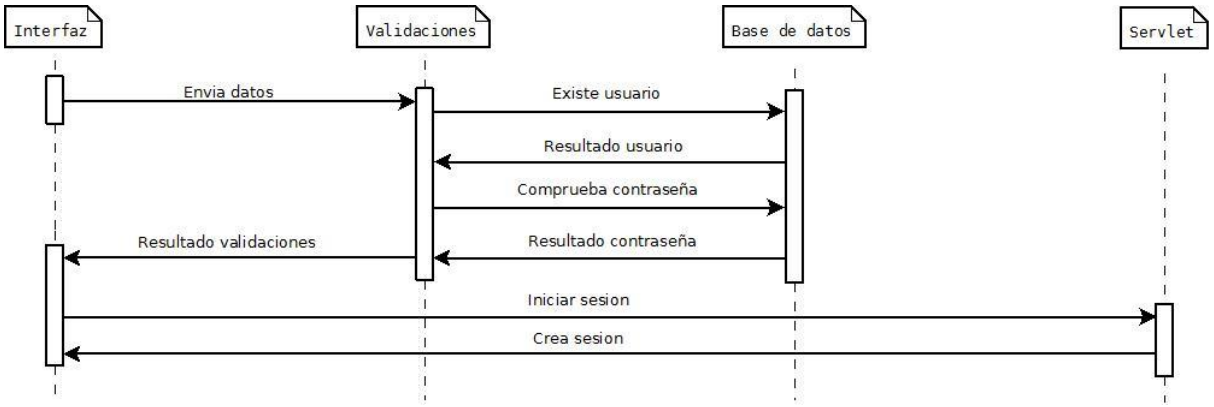


Ilustración 9: DS Iniciar sesión

- Iniciar búsqueda de *tuits*.

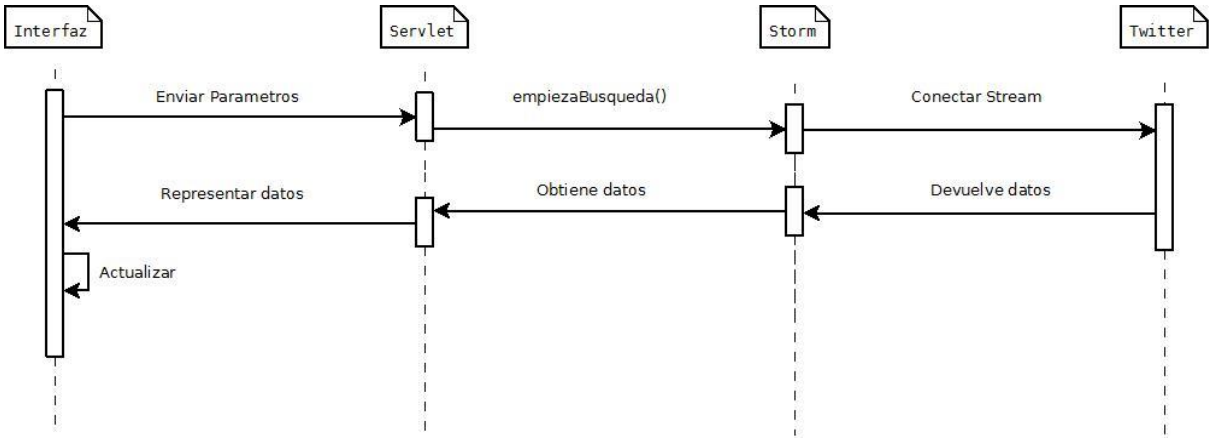


Ilustración 10: DS Búsqueda

- Ver *tuits* que se han registrado en cada región.

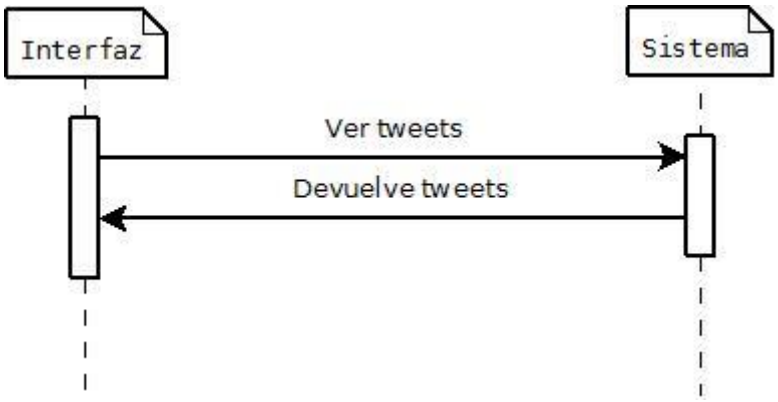


Ilustración 11: DS Ver tuits

- Ver localizaciones totales obtenidas al realizar la búsqueda.

Ilustración 12: DS Ver localizaciones

- Guardar resultados obtenidos de las búsquedas realizadas.

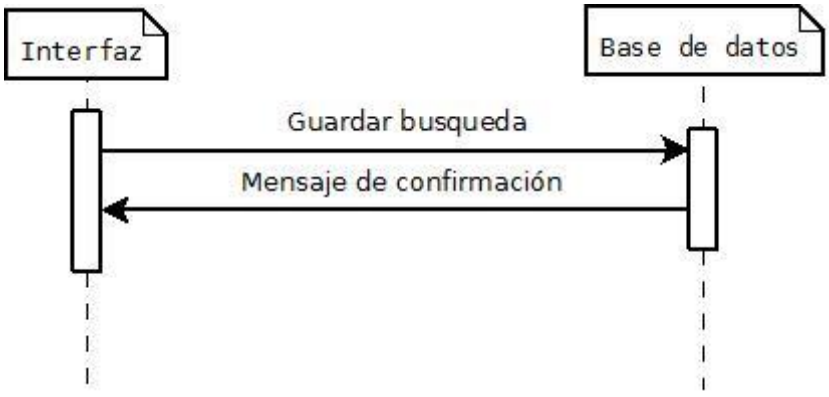


Ilustración 13: DS Guardar búsqueda

- Ver perfil de usuario actual.

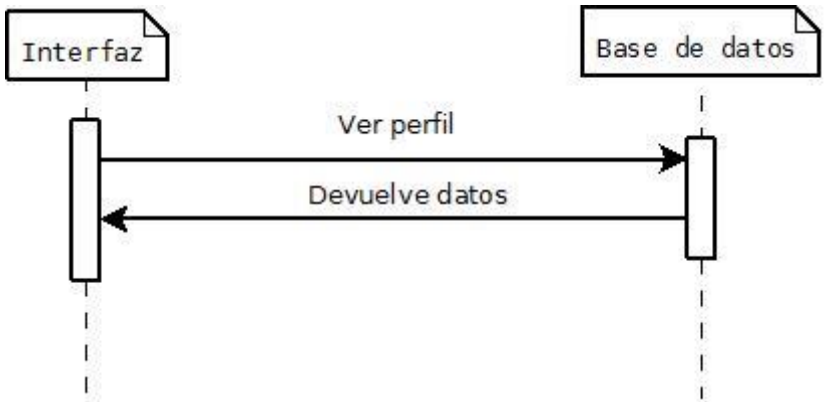


Ilustración 14: DS Ver perfil

- Cambiar contraseña actual del usuario.

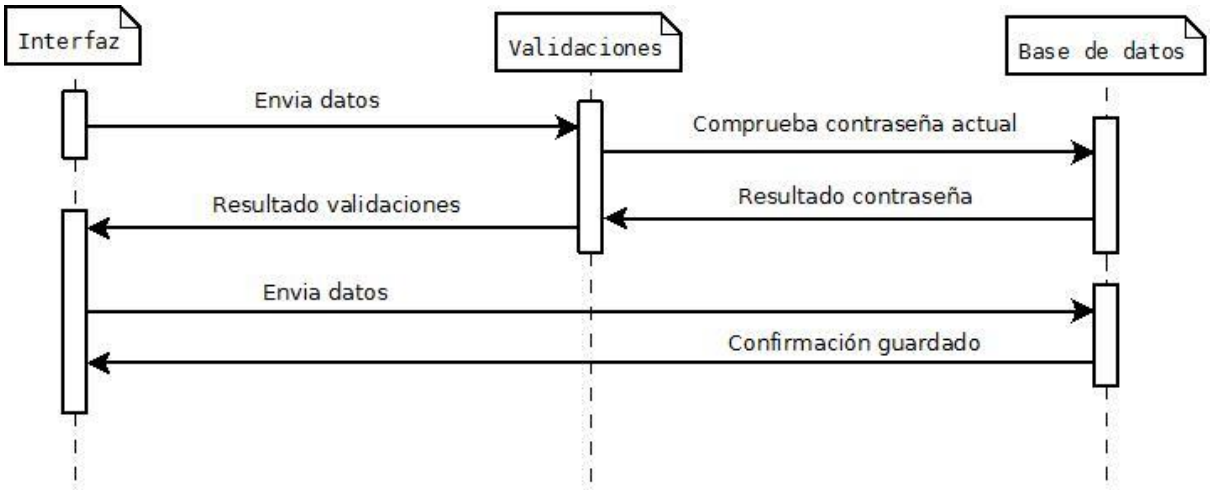


Ilustración 15: DS Cambiar contraseña

- Cerrar sesión del usuario actual.

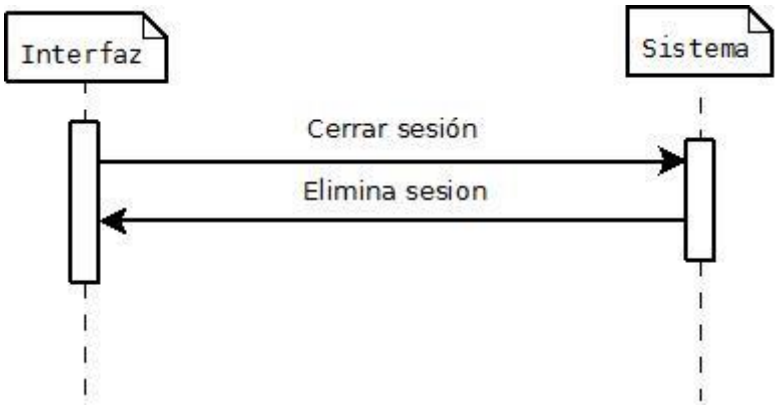


Ilustración 16: DS Cerrar sesión

6.3.2. Diagramas de secuencia relacionados con el sistema

En esta sección se muestran los diagramas de secuencia de las operaciones en las que interfiere el sistema:

- Conectar con el *stream* de twitter.

Ilustración 17: DS Conectar a Twitter

- Obtener la ubicación exacta de un *tuit* mediante *OpenStreetMap*.

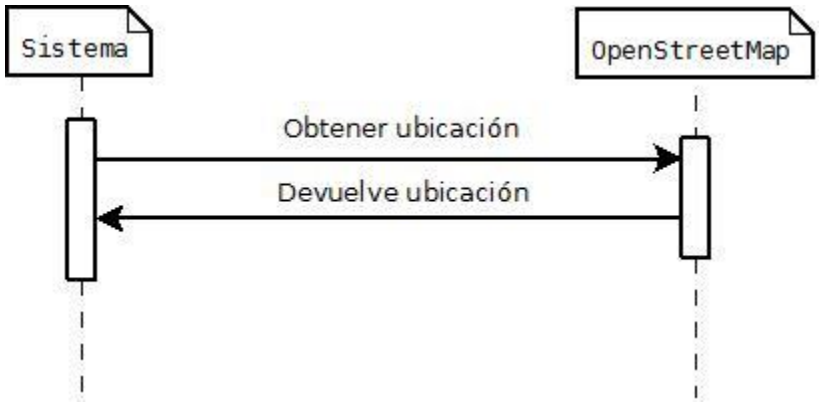


Ilustración 18: DS Obtener ubicación

7. Diseño

En este capítulo se detallarán los aspectos más específicos acerca del diseño que se ha implementado en el sistema.

7.1. Arquitectura del sistema

En esta sección se muestra la arquitectura que compone el sistema. A continuación, en la *ilustración 19* se puede observar los componentes que forman la arquitectura de la aplicación.

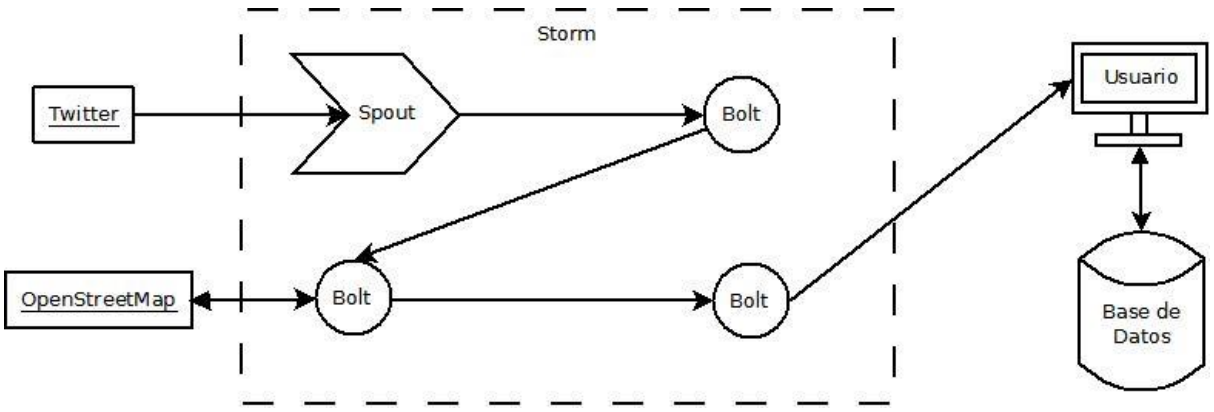


Ilustración 19: Arquitectura Storm

Lo que se puede apreciar en la *ilustración* 19, es la esencia de la aplicación, el motor del sistema, es la topología de *storm*. Gracias a esta topología se puede conectar con Twitter, obtener los *tuits* que se generan en ese instante en tiempo real, procesarlos, obtener los datos que nos interesan y representarlos gráficamente en un mapa en relación a los datos de ubicación que se obtenga de ellos.

Este es un proceso complicado en el que los componentes de la topología deben estar perfectamente conectados y sincronizados para que los datos vayan siempre relacionados entre sí y no se mezclen, por ejemplo, la ubicación de un *tuit* con el mensaje de otro distinto.

El primer paso de la topología es conectar el *stream* de Twitter, con el *spout* de *storm*. Para ello, twitter4j proporciona una clase llamada ***TwitterStream***. Ésta clase, pide que se le proporcione las credenciales de acceso a Twitter (*TwitterKeys*, se especifica en el Manual de de desarrollador) y un *listener*. El *listener* simplemente se encarga de obtener los datos que se vayan generando, en este caso de Twitter.

Una vez realizado este proceso, el segundo paso es mandarle al *stream* de Twitter una consulta personalizada, en este caso, una palabra para filtrar, es decir, obtener los *tuits* que se vayan generando que contengan la palabra o conjunto de palabras especificadas. Al igual que filtrar por palabras, hay multitud de consultas que se pueden realizar. Otra consulta realizada en la topología es, filtrar todos los *tuits* que se generen en un área geográfica concreta, en este caso España. El área geográfica se especifica proporcionando las coordenadas del *Bounding Box* [2] deseado, que consiste en un área cuadrada en el mapa.

Proporcionados estos elementos, obtenemos un *stream* de datos personalizado de Twitter que nos proporcionará sólo los *tuits* que cumplan estas características.

A partir de aquí, entra en juego el ***spout*** de *storm*. El *spout* se encargará de tomar esos datos proporcionados por el *stream* de Twitter, crear otro *stream* interno a la topología, y emitir los datos obtenidos por ese *stream*.

Con esto, ya tenemos la posibilidad de almacenar los *tuits*, el inconveniente está en que la información se obtiene en formato Json, además de que nos proporciona muchos más datos de los que necesitamos como, fecha de creación de la cuenta de Twitter o el número de seguidores entre otros.

Para realizar el procesado de la información, se utilizarán los ***bolts***. Como se puede ver en la *ilustración 19*, esta topología dispone de tres *bolts*, cada uno con una función distinta que se detallará a continuación.

El *spout* de la topología, ya ha creado el *stream* por el que va a emitir los datos que obtiene de Twitter. Este *stream* seguirá su camino y se conectará al primer *bolt*.

En este primero *bolt*, se realiza el proceso de extracción de los datos que verdaderamente nos interesan de todos los que nos proporciona el *tuit* capturado. Para esto, *twitter4j* cuenta con métodos capaces de extraer cada campo de información existente.

En primer lugar, se extrae el nombre de usuario del autor del *tuit*. En segundo lugar, se extrae el mensaje que contiene el *tuit*. Y por último, se extrae la ubicación del *tuit*.

Para extraer los dos primeros campo se utiliza una función, pero para la ubicación hace falta un proceso un poco más tedioso debido a que se pueden obtener más de una ubicación.

Las distintas ubicaciones que se pueden obtener son:

- El nombre de la ubicación desde la que se ha escrito el *tuit*.
- Las coordenadas de geolocalización de la ubicación desde la que se ha escrito el *tuit*.
- El nombre de la ubicación que tiene especificada el usuario en su perfil de Twitter.

Sabiendo esto, el proceso de obtención de ubicación se realiza de la siguiente forma. En primer lugar se busca el nombre de la ubicación del *tuit*, si este no existe, se buscan las coordenadas desde las que se ha escrito el *tuit,* y si tampoco existiera, se obtiene la información de la ubicación del perfil de usuario de Twitter.

Una vez obtenidos todos estos datos, se forma un String con el usuario y el mensaje del *tuit*, y otro String con la ubicación obtenida. Los dos Strings se emiten en el *stream* de datos para que los reciba el segundo *bolt*.

El *stream* sigue su camino y transfiere los datos desde el primer *bolt* al segundo. En este **segundo *botl***, se realiza el proceso de obtener una ubicación válida del *tuit*. En primer lugar, el *bolt* recibe los datos enviados por el *stream*, es decir, dos cadenas que contienen el usuario y el mensaje del *tuit* en una, y la ubicación en otra. El proceso consiste en obtener el valor de la ubicación y contrastarla con *OpenStreetMap* consultando una *url* que proporciona la plataforma. Una vez consultada, la *url* emite un Json con todos los datos de la ubicación como el país al que pertenece, las coordenadas, o si es una isla, entre otros.

Una vez obtenido ese Json, lo procesamos y obtenemos la información que no interesa en un proceso que consiste en, primero, averiguar si la ubicación existe; segundo, averiguar si la ubicación pertenece a España; y por último, obtener la provincia o la comunidad autónoma a la que pertenece la ubicación dependiendo de lo que haya decidido el usuario del sistema.

Si la ubicación no pertenece a España, se le asigna el nombre “Extranjero” y si la ubicación no existe, se le asigna el nombre “Sin ubicación”.

Una vez obtenido el nombre exacto de la ubicación, se almacena en una primera lista de ubicaciones el nombre y el número de veces que se ha obtenido esa misma ubicación a lo largo de la búsqueda realizada.

Cuando se termina este proceso, se emite el valor de la ubicación y el usuario con el mensaje del *tuit* en el *stream* de datos.

El *stream* sigue su procedimiento y transfiere los datos al tercer y **último *bolt***. Este *bolt* se encargará de agrupar todos los *tuits* diferenciándolos por su ubicación y lo hará de la siguiente forma.

El *bolt* obtiene los datos del segundo *bolt* y empieza a procesarlos. En primer lugar obtiene la localización, y comprueba que no tenga el valor “Extranjero” o “Sin ubicación”, pues estos no se representarán en el mapa. Si la localización no tiene ninguno de esos dos valores, se accede a una segunda lista de ubicaciones y se comprueba si esa localización ya está guardada en la lista por un *tuit* procesado anteriormente en la ejecución. Si la localización no existe, se añade a la lista y a su vez se le asigna una lista de *tuits* que se han registrado en esa ubicación, de forma que cada ubicación estará relacionada con una lista de *tuits* generados en esa ubicación; si la ubicación ya existe en la lista, se añade el mensaje y el usuario a la lista que está relacionada con esa ubicación. Quedaría estructurado tal y como se ve en la *ilustración 20:*

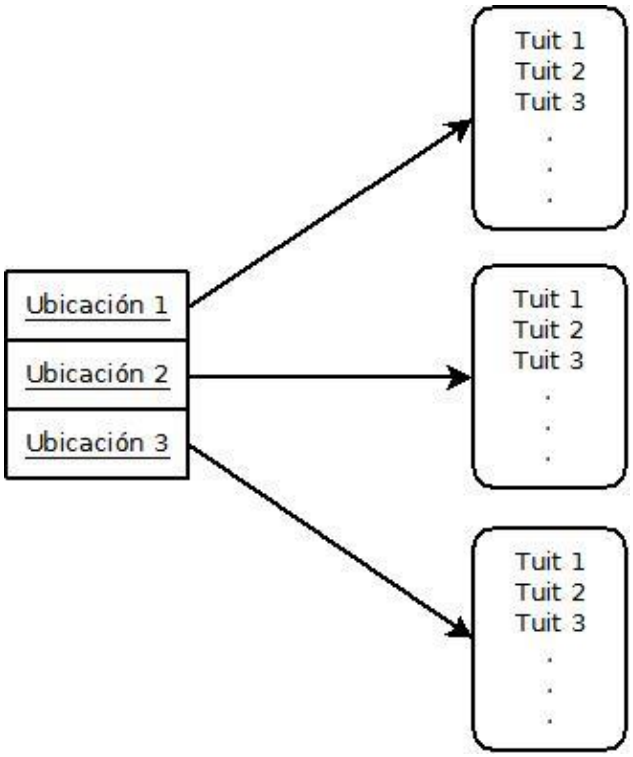


Ilustración 20: Lista de ubicaciones-tuits

Una vez realizado el proceso, el usuario puede obtener estas listas mostradas por pantalla pulsando los diferentes botones proporcionados para ello.

7.2. Diseño físico de datos

En esta sección se detallarán las estructuras de datos que maneja la aplicación. A continuación, se muestra, a base de tablas, las clases de datos que forman la aplicación.

Usuario		
Nombre	**Descripción**	**Tipo**
Id	Identifica mediante un entero a cada usuario. El id es único.	Integer
Nombre	Nombre del usuario	String
Apellidos	Apellidos del usuario	String
Usuario	Nombre de usuario en la aplicación. Este campo es único.	String
Contraseña	Contraseña del usuario para acceder a la aplicación.	String

Tabla 2: Diseño físico de datos. Usuario

Búsqueda		
Nombre	**Descripción**	**Tipo**
Id	Identifica con un entero cada búsqueda almacenada. Es único.	Integer
Usuario	Id del usuario que ha almacenado la búsqueda	Usuario
Fecha	Fecha en la que se guarda la búsqueda.	Date
Filtro	Filtro que se ha utilizado como parámetro de búsqueda.	String
Zoom	Zoom que se ha utilizado como parámetro de búsqueda.	String
Tiempo	Tiempo que se ha utilizado como parámetro de búsqueda.	String
Tuits	Número de *tuits* totales que se han obtenido en la búsqueda.	String
Localizaciones	Número de localizaciones que se han obtenido en la búsqueda.	String

Tabla 3: Diseño físico de datos. Búsqueda

Estos modelos de datos, son los que se usarán para crear las tablas de la base de datos. Cada tabla tendrá una fila por cada atributo que tiene el modelo de datos. En este caso, la aplicación cuenta con dos modelos de datos, puesto que solo posee dos tablas relacionadas entre sí, **Usuario** y **Búsqueda**, con una relación de **uno a muchos**: un usuario, múltiples búsquedas.

De los atributos de la entidad **Usuario**, cabe destacar el **Id**, que, mediante hibernate, está definido como único en base de datos, por lo que no se pueden almacenar dos iguales. Además, hibernate también se encarga de asignarle un Id automáticamente a cada instancia de la clase por lo que provoca que este atributo no pueda ser nulo nunca.

En la entidad **Busqueda**, además del atributo Id, que tienen exactamente la misma función que en la entidad Usuario, también podemos encontrar otros dos atributos destacables como son, **Usuario** y **Fecha**.

El atributo **Usuario** es una instancia de la clase Usuario y sirve para relacionar una búsqueda con un usuario existente en la base de datos.

El atributo **Fecha** es una instancia de la clase Date, este almacena una fecha con el formato aaaa-MM-dd hh:mm:ss que se crea automáticamente al crear la instancia obteniendo la hora del sistema.

8. Implementación del sistema

En este capítulo se mostrarán todos los aspectos relacionados con la implementación del sistema.

8.1. Entorno de construcción

En esta sección se detallarán las herramientas utilizadas para desarrollar el código fuente de la aplicación, así como las librerías que se han necesitado y el software necesario para la correcta gestión del desarrollo del proyecto.

8.1.1. Eclipse

Como *Entorno de Desarrollo Integrado* se ha optado por elegir la opción de *Eclipse*. La principal causa de esta elección es la experiencia de uso que ya se ha obtenido antes de comenzar con el desarrollo de este proyecto. Además *Eclipse* incluye numerosas herramientas y *plugins* que se integran perfectamente con el entorno de desarrollo y facilitan la gestión y realización del proyecto.

La versión de *Eclipse* escogida en esta ocasión es, *Eclipse Neon* en su versión *4.6.1*. Se ha elegido esta versión principalmente porque es una versión con la que se había trabajado antes y ha dado buenos resultados.

Ilustración 21: Versión Eclipse

Además, se ha usado la versión de java 8 como lenguaje de programación.

8.1.2. Maven

En lo que respecta a la obtención de librerías necesarias para el funcionamiento del proyecto, se ha optado por usar *Maven*. Esta herramienta de gestión para la construcción de software, facilita y agiliza el proceso de inclusión de librerías en el proyecto.

Maven dispone de un archivo *POM.xml* en el que se incluyen las dependencias que se necesitan para el desarrollo y a la hora de construir el proyecto, *Maven* se encarga de descargar todas las librerías e incluirlas automáticamente.

Así, las dependencias que se han incluido en el archivo *POM.xml* de *Maven* son las siguientes:

```
<dependencies>
  <dependency>
      <groupId>org.apache.storm</groupId>
      <artifactId>storm-core</artifactId>
      <version>0.9.1-incubating</version>
  </dependency>
  <dependency>
      <groupId>org.twitter4j</groupId>
      <artifactId>twitter4j-stream</artifactId>
      <version>4.0.4</version>
  </dependency>
  <dependency>
      <groupId>org.json</groupId>
      <artifactId>json</artifactId>
      <version>20160810</version>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
  </dependency>
  <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>5.2.2.Final</version>
  </dependency>
  <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.37</version>
  </dependency>
</dependencies>
```

Ilustración 22: Dependencias Maven

8.1.3. Git

Para el control de versiones del proyecto, se ha utilizado la herramienta *Git*. Para usarlo, se ha instalado el *plugin* correspondiente en eclipse y se ha configurado el repositorio en el que se van almacenando todas las versiones que van surgiendo del proyecto.

En este caso, el proyecto está almacenado en la url: https://neptuno.uca.es/git/tfg-antonio-rodas, que se trata de un repositorio de la plataforma *Neptuno*.

8.1.4. Apache Tomcat

Como contenedor de *Servlets*, se ha utilizado Apache Tomcat v8.0. El proyecto se ha desarrollado usando este contenedor pero siempre desplegando la aplicación en localhost.

8.2. Base de datos

La base de datos elegida para el proyecto es el sistema de gestión *MySQL*. Se ha elegido este sistema debido a que es uno de los más populares del mercado, y además, ya se han tenido experiencias anteriores en el uso de este sistema.

Para habilitar la interacción entre la aplicación y la base de datos, se ha usado **hibernate**.

Hibernate es una herramienta que permite el mapeo de atributos entre una base de datos relacional y el modelo de datos de una aplicación.

Al usar *hibernate*, no es necesario usar ningún *script* de creación de base de datos ya que el propio *hibernate*, al desplegar el proyecto en el servidor, se encarga de crear la base de datos, y todas las tablas pertinentes, si ésta no existe, consiguiendo así un proceso completamente automatizado y mucho más fácil de realizar.

A continuación se detallará cómo se realiza la interacción de la aplicación con la base de datos donde se almacenan los usuarios registrados y los resultados de las búsquedas realizadas que los usuarios hayan decidido guardar.

Para poder usar hibernate, se necesita un archivo de configuración con nombre: **hibernate.cfg.xml.** En este archivo se declararán las configuraciones pertinentes.

8.2.1. Anotaciones hibernate

En primer lugar, deberemos incluir en el archivo *POM.xml*, las dependencias necesarias, que en esta ocasión son la dependencia de *hibernate* y la dependencia del *conector MySQL*, de la misma forma que se puede observar en la *ilustración 22*.

Para realizar el mapeo de datos, se necesitan añadir una serie de anotaciones a los tipos de datos que se van a almacenar en la base de datos. En este caso los tipos de datos que se quieren almacenar son: **Usuario** y **Busqueda**. A continuación se muestran las anotaciones que son necesarias añadir a las clases especificadas.

- **Usuario.java**

```
@Entity
@Table(name="Usuarios")
public class Usuario {

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	@Column(name="Id")
	private int id;

	@Column(name="Nombre")
	private String nombre;

	@Column(name="Apellidos")
	private String apellidos;

	@Column(name="Usuario")
	private String usuario;

	@Column(name="Contrasena")
	private String contrasena;
```

- **Busqueda.java**

```
@Entity
@Table(name="Busquedas")
public class Busqueda {

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	@Column(name="Id")
	private int id;

	@ManyToOne
	@JoinColumn(name="Usuario")
	private Usuario usuario;

	@Column(name="Fecha")
	private Date fecha;
```

```
@Column(name="Filtro")
private String filtro;

@Column(name="Zoom")
private String zoom;

@Column(name="Tiempo")
private String tiempo;

@Column(name="Tuits")
private String tuits;

@Column(name="Localizaciones")
private String localizaciones;
```

Como se puede observar en el código, todas las anotaciones se empiezan por "@" y pertenecen a la clase *javax.persistence*.

Al principio del código tenemos la anotación *@Entity*, que se encarga de crear una tabla en la base de datos que se corresponde con el tipo de datos de la clase. Esta tabla tendrá el nombre que se especifica con la anotación *@Table*. A partir de aquí, se especifican las columnas que contendrá la tabla añadiendo la anotación *@Column* a cada atributo de la clase.

Como se puede observar, hay dos anotaciones más que se añaden al atributo *id* y dos más que se añade al atributo *usuario* de tipo *Usuario*.

La anotación *@Id* se encarga de establecer el atributo *id* de la clase como el *id* de los objetos almacenados en la base de datos, estableciendo este atributo como **único** y **no nulo**. A continuación se especifica la anotación *@GeneratedValue(strategy=generatioType.IDENTITY)*, la cual se encarga de generar automáticamente un valor para el atributo *id* de la clase siguiendo una secuencia.

Por último, en el atributo *usuario* de tipo *Usuario*, se puede observar la anotación *@ManyToOne* que es usada para establecer una relación de *muchos a uno*. Esto significa que establecemos una relación en la que, una *búsqueda*, tiene un solo *usuario* y un *usuario* puede tener *n busquedas*. La anotación *@JoinColumn* simplemente sirve para darle un nombre a esa columna en la base de datos.

8.2.2. Configuración hibernate

Para concluir el proceso, se configura el archivo **hibernate.cfg.xml** de la siguiente forma:

- **hibernate.cfg.xml**

```
<hibernate-configuration>
   <session-factory>

     <property name="hibernate.dialect">
           org.hibernate.dialect.MySQLDialect
      </property>

     <property name="hibernate.connection.driver_class">
           com.mysql.jdbc.Driver
      </property>

     <property name="hibernate.connection.username">
           [usuario - base de datos]
      </property>

     <property name="hibernate.connection.password">
           [contraseña - base de datos]
      </property>

      <property name="hibernate.connection.url">
jdbc:mysql://[url - base de datos]?createDatabaseIfNotExist=true
      </property>

     <property name="connection_pool_size">1</property>

     <property name="hbm2ddl.auto">update</property>

     <property name="show_sql">true</property>

     <mapping class="[claseUsuario]"/>
     <mapping class="[claseBusqueda]"/>

   </session-factory>
</hibernate-configuration>
```

En este archivo, habrá que especificar los siguientes campos:

- ***hibernate.connection.username:*** El usuario para acceder a la base de datos.
- ***hibernate.connection.password*****:** La contraseña para acceder a la base de datos.
- ***hibernate.connection.url:*** Habrá que especificar la url donde se encuentra la base de datos de esta forma:
    - ***jdbc:mysql://*** **:** Trivial para conexiones a bases de datos *MySQL*.
    - ***[url]*****:** url de la base de datos.
    - ***?createDatabaseIfNotExist=true*****:** Esta sentencia posibilita que se cree la base de datos automáticamente si ésta no existe.

Quedando una sentencia final de esta estructura:
jdbc:mysql://localhost:3306/twitterstorm?createDatabaseIfNotExist=true

- ***connection_pool_size*:** Número de hilos de conexión a la base de datos.
- ***hbm2ddl.auto*:** Aqui se especifica la forma en que se va a tratar la base de datos:
  - ***validate:*** Valida el esquema de base de datos. No realiza ningún cambio.
  - ***create:*** Crea el esquema nuevo de base de datos y si hay uno ya creado, lo borra y crea uno nuevo.
  - ***update:*** Crea el esquema de base de datos, solo si no existe un esquema ya creado.
  - ***create-drop:*** Crea el esquema nuevo de base de datos y si existe uno, lo borra y crea uno nuevo. Y además, borra el esquema cuando se cierra la sesión. Esta opción es usada principalmente para pruebas de desarrollo.
- ***show_sql:*** Muestra el *sql* ejecutado en cada acción por consola. Simplemente utilizado para llevar un seguimiento de las operaciones realizadas.
- ***<mapping>:*** Se deben indicar las clases del proyecto que se quiere que sean mapeadas por hibernate.

Los demás campos son triviales para la conexión a una base de datos *MySQL*.

Finalmente, se crean las clases **DAO** que se encargan de realizar las operaciones de gestión de datos contra la base de datos.

8.2.3. Acceso a datos hibernate

Para realizar operaciones de gestión contra la base de datos, se crea una clase *DAO* por cada entidad mapeada del proyecto, con el fin de abstraer las operaciones y facilitar la interpretación del código.

A continuación de detallará el comportamiento de la clase ***busquedaDAO.java*** ya que las funciones son las mismas que *usuarioDAO.java* pero añadiendo alguna función más.

busquedaDAO.java

```
public class BusquedaDao {

	private Session sesion;
	private Transaction transaction;

	 /**
	* Guardar una Busqueda en la base de datos
	* @param busqueda Instancia de la clase Busqueda
```

```
     * @return Id de la instancia almacenada del objeto Busqueda
     * @throws HibernateException Excepci&oacute;n de hibernate
     */
     public int saveBusqueda(Busqueda busqueda) throws
HibernateException {

          int id = 0;

          try{

               iniciaSesion();
               id = (int)sesion.save(busqueda);
               transaction.commit();

          } catch (HibernateException he) {
               manejaExcepcion(he);
               throw he;
          } finally {
               sesion.close();
          }

          return id;
     }

     /**
     * Actualiza una Busqueda que ya existen en la base de datos
     * @param busqueda Instancia de la clase Busqueda
     * @throws HibernateException Excepci&oacute;n de hibernate
     */
     public void updateBusqueda(Busqueda busqueda) throws
HibernateException {

          try{
               iniciaSesion();
               sesion.update(busqueda);
               transaction.commit();
          } catch (HibernateException he) {
               manejaExcepcion(he);
               throw he;
          } finally {
               sesion.close();
          }
     }

     /**
     * Borrar Busqueda que se encuantra en la base de datos
     * @param busqueda Instancia de la clase Busqueda
     * @throws HibernateException Excepci&oacute;n de hibernate
     */
     public void deleteBusqueda(Busqueda busqueda) throws
HibernateException {

          try{
               iniciaSesion();
               sesion.delete(busqueda);
               transaction.commit();
```

```
            } catch (HibernateException he) {
                manejaExcepcion(he);
                throw he;
            } finally {
                sesion.close();
            }
      }

      /**
      * Obtener Busqueda de la base de datos mediante su Id.
      * @param idBusqueda Id de la instancia de la clase Busqueda
      * @return Instancia de la clase Busqueda o null si el id no
existe en la base de datos
      * @throws HibernateException Excepci&oacute;n de hibernate
      */
      public Busqueda getBusqueda(int idBusqueda) throws
HibernateException {

            Busqueda busqueda = null;

            try{
                iniciaSesion();
                busqueda = sesion.get(Busqueda.class,
idBusqueda);
            } finally {
                sesion.close();
            }

            return busqueda;
      }

      /**
      * Obtiene todas las Bsuqedas almacenadas en la base de
datos
      * @return Una Lista con todas las instancias de la clase
Busqueda de la base de datos
      * @throws HibernateException Excepci&oacute;n de hibernate
      */
      @SuppressWarnings("unchecked")
      public List<Busqueda> getBusquedas() throws
HibernateException {

            List<Busqueda> busquedas = null;

            try{
                iniciaSesion();
                busquedas = sesion.createQuery("from
Busqueda").getResultList();
            } finally {
                sesion.close();
            }

            return busquedas;
      }

      /**
```

```
	 * Obtiene todas la busquedas almacenadas por un usuario
especificado
	 * @param usuario Una instancia de la clase Usuario
	 * @return Una lista con todas las instacias de la clase
Busqueda que esten relacionadas
	 * con el usuario especificado
	 * @throws HibernateException Excepci&oacute;n de hibernate
	 */
	@SuppressWarnings("unchecked")
	public List<Busqueda> getBusquedasByUsuario(Usuario
usuario) throws HibernateException {

		List<Busqueda> busquedas = null;

		try{
			iniciaSesion();
			busquedas = sesion.createQuery("from Busqueda
where usuario = '"+usuario.getId()+"'").getResultList();
		} finally {
			sesion.close();
		}

		return busquedas;
	}

	 /**
	 * Crea una sesion para conectarse a la base de datos
	 * @throws HibernateException Excepci&oacute;n de hibernate
	 */
	private void iniciaSesion() throws HibernateException {
	  this.sesion = SessionFactoryUtil.openSession();
	    this.transaction = this.sesion.beginTransaction();
	}

	 /**
	  * Maneja las execpciones que puedan ocurrir durante el
acceso a la base de datos
	 * @param HibernateException Excepci&oacute;n de hibernate
	 * @throws HibernateException Excepci&oacute;n de hibernate
	 */
	private void manejaExcepcion(HibernateException he) throws
HibernateException {
		transaction.rollback();
		throw new HibernateException("Error accediendo a los
datos", he);
	}
}
```

La clase se compone de las siguientes funciones:

- **iniciaSesion():** Se encarga de crear una sesión para conectarse con la base de datos y comenzar la transacción de información.

- **manejaExcepcion(Excepcion):** Se encarga de controlar la excepción y evitar que se produzcan incoherencias en la base de datos.

- **savebusqueda(Busqueda):** Se encarga de almacenar un objeto *Busqueda* en la base de datos, guardando cada atributo en la columna correspondiente y asignándole un id automáticamente. Crea la sesión, almacena el objeto, realiza el *commit* en base de datos y cierra la sesión.

- **updatebusqueda(Busqueda):** Se encarga de actualizar la información de una búsqueda que ya existe en la base de datos. Crea la sesión, actualiza la búsqueda que coincide con el id del objeto, realiza el *commit* en base de datos y cierra la sesión.

- **deleteBusqueda(Busqueda):** Se encarga de eliminar una búsqueda que ya existe en la base de datos. Crea la sesión, elimina la *Busqueda* que coincide con el id del objeto, realiza el *commit* en base de datos y cierra la sesión.

- **getBusqueda(idBusqueda):** Obtiene un objeto *Busqueda* buscando un id en la base de datos. Crea la sesión, obtiene la *Busqueda* mediante el id, cierra la sesión y devuelve el objeto *Busqueda.*

- **getBusquedas():** Obtiene todas las *Busqueda* almacenadas en la base de datos. Crea la sesión, obtiene todas las *Busqueda* y las almacena en un *ArrayList*, cierra la sesión y devuelve la lista.

- **getBusquedaByUsuario(Usuario):** Obtiene una lista de *Busqueda* que haya almacenado un *Usuario* concreto. Crea la sesión, hace una consulta de las *Busqueda* que haya almacenado un *Usuario* y las almacena en un *ArrayList*, cierra la sesión y devuelve la lista.

8.3. Código fuente

En esta sección se explicara el funcionamiento del sistema a nivel de código fuente dividiéndolo en dos perspectivas: implementación de *storm* e implementación de la interfaz.

8.3.1. Implementación de storm

En esta sección se detallara el código usado para implementar los distintos componentes que forman la topología *storm* del sistema que se puede ver en la *ilustración 19*.

- **TwitterSpout.java**

En esta clase, se implementa el *spout* de la topología, que se encarga de obtener los datos que proporciona Twitter. A continuación se muestra la parte más importante de la clase:

```
ConfigurationBuilder cb = new ConfigurationBuilder();
cb.setDebugEnabled(true)
      //Se establecen las credenciales de twitter
      .setOAuthConsumerKey(consumerKey)
      .setOAuthConsumerSecret(consumerSecret)
      .setOAuthAccessToken(accessToken)
      .setOAuthAccessTokenSecret(accessTokenSecret);

_twitterStream = new
TwitterStreamFactory(cb.build()).getInstance();
//Se crea la instancia del stream de twitter
_twitterStream.addListener(listener);
//Se añade el capturador de informacion
FilterQuery query = new FilterQuery();
//Se crea una instancia de la consulta
if((keyWords == null) || (keyWords.length == 0)){
//Si no se establecen filtros
      double[][] Spain =
{{-18.45703125,27.3815231917},{4.6362304687,43.953282042}};
//Se establecen las coordenadas del area geografica donde se
quiere buscar
      query.locations(Spain);
      //Se establece una busqueda por area geografica
      _twitterStream.filter(query);
      //Se establecen los parametros de la consulta
}else{
      //Si el filtro no esta vacio
      query.track(keyWords);
      //Se establece una busqueda por palabras
      _twitterStream.filter(query);
      //Se establecen los parametros de la consulta
   }
```

En primer lugar, se crea una configuración a la que se le añaden las credenciales de Twitter, las TwitterKeys. A continuación, se crea una instancia de la clase TwitterStream, proporcionada por la librería *twitter4j* que sirve para obtener el *stream* de Twitter. Se inicializa con la instancia de la configuración y luego se le añade un capturador de información que se encarga de esperar a que se produzcan datos para obtenerlos.

El segundo paso es crear la consulta que va a parametrizar el *stream* de Twitter. Para esto, se crea una *query* a la que se le añadirá los parámetros específicos. La *query* se le añade al *stream* por medio de la función *filter()*, la cual, tiene el inconveniente de que no permite anidar las consultas por lo que se ha tomado la siguiente solución. Antes de realizar la consulta, se comprueba la lista de palabras que se van a usar para filtrar los *tuits*; si esta lista esta vacía o la palabra está en blanco, se establecerá otro tipo de consulta mediante la función *locations()* que consistirá en obtener los *tuits* generados en un área geográfica determinada; para ello, se establecen las coordenadas del *Bounding Box* [2]

que contiene, en este caso, al área geográfica de España. Una vez establecidas las coordenadas, se realiza el filtrado, obteniendo todos los *tuits* que se generan en España.

Por otra parte, si en la lista de palabras está llena, se omitiría la búsqueda por coordenadas estableciendo una búsqueda por palabras mediante la función *track()*. A esta función se le pasa el array de palabras que se han establecido como parámetros. Si la palabra es una, el array solo contendrá una posición. A continuación, se realiza el filtrado del *stream*, obteniendo así todos los *tuits* que contengan la palabra especificada en la *query*.

```
public void nextTuple(){
	Status ret = queue.poll();
	if(ret != null){//Si el valor obtenido no es null
		_collector.emit(new Values(ret));
		//Se emite en el stream de la topologia
	}
}
```

Una vez obtenido el *stream* de Twitter parametrizado, mediante a función *nextTuple()*, se crea un *stream* de storm y se emiten los *tuits* obtenidos para que los reciba el próximo componente de la topología.

- **TwitterReaderBolt.java**

Una vez ejecutado el *spout*, ya se obtienen los datos de los *tuits* que estamos buscando, el problema está en que los *tuits* traen demasiados datos y muchos no los vamos a usar, además, vienen en formato Json, por lo que habrá que procesarlo y obtener solo los datos que necesitamos y en el formato en el que los necesitamos. Para ello se crea esta clase, el primer *bolt* de la topología, que es el paso siguiente al *spout* y por tanto recibe de ahí los datos a procesar.

```
public void execute(Tuple tuple){

	final Status tweet = (Status) tuple.getValueByField("tweet");
	String location = "Vacio";
	String message = "<a href=\"https://twitter.com/"+
	tweet.getUser().getScreenName()+"\"
	target=\"_blank\">@"+tweet.getUser().getScreenName()+"</a> dijo:
	"+tweet.getText();
	//obtiene el nombre de usuario y el texto del tweet
	double latitude;
	double longitude;
	Place place = tweet.getPlace();

	if(place != null){
		location = place.getFullName();
		//obtiene el lugar del que proviene el tweet
	} else {
		if(tweet.getGeoLocation() != null){
			latitude = tweet.getGeoLocation().getLatitude();
			//obtiene latitud del tweet
			longitude = tweet.getGeoLocation().getLongitude();
			//obtiene longitud del tweet
			location = latitude +","+longitude;
```

```
            } else {
                  if(tweet.getUser().getLocation() != null){
                        location = tweet.getUser().getLocation();
                        //obtiene la localizacion del usuario
                        }
                  }

            }


            this.collector.emit(new Values(location,message));
             //Emite el resultado al stream
      }
```

La función de este *bolt* es obtener los datos que realmente necesitamos para hacer funcionar la aplicación, como son: el usuario que escribió el *tuit*, el mensaje que escribió, y la ubicación del *tuit*.

Los dos primeros son mas sencillos de conseguir, ya que el usuario se obtienen mediante la función *getUser().getScreenName()* y el mensaje mediante *getText()*.

La cosa se complica cuando hay que obtener la ubicación que, como se explica en capítulos anteriores, se pueden obtener tres ubicaciones distintas. En el código se muestra como obtenerlas las tres.

En primer lugar se comprueba el nombre de la localización de la que proviene el *tuit* mediante la función *getPlace().getFullName()*. Si el atributo *Place* del *tuit* es nulo, se comprueba si el *tuit* dispone de geolocalización y se obtienen las coordenadas de la ubicación mediante las funciones *getGeoLocation().getLatitude()* y *getGeoLocation().getLongitude()*. Si el atributo de geolocalización del *tuit* también fuera nullo, entonces, como última opción, se obtendría la ubicación que ha especificado el usuario en su perfil de Twitter mediante la función *getUser().getLocation()*.

Si todos los atributos fueran nulos, entonces se asignaría al *tuit* el valor "Sin ubicación".

Una vez dado al mensaje el formato deseado y obtenida la ubicación del *tuit*, se emiten los resultados en el *stream* de *storm* para que los datos puedan ser obtenidos por el siguiente componente.

- **TwitterWriteBolt.java**

El siguiente *bolt* que recibe los datos del *stream* se encarga de obtener una ubicación exacta del *tuit* consultando una *url* de *OpenStreetMap*. A continuación se muestra cómo.

```
public void execute(Tuple tuple) {
            String location = tuple.getString(0);
            //obtiene el primer campo del Stream -> localizacion
            String message = tuple.getString(1);
            //obtiene el segundo campo del Stream -> usuario
      dijo: texto
            JSONObject json = null;
            if(!(location.equals("Vacio"))){
```

```
                try {
                      location = URLEncoder.encode(location,
"UTF-8");
                      //obtiene un formato de String valido para
                la url

                      //Consulta la localizacion y obtiene un
json
                      json =
j.readJsonFromUrl("http://nominatim.openstreetmap.org/search?q="
+location+"&format=json&limit=1&accept-language=es");
                } catch (IOException | JSONException e){
                      e.printStackTrace();
                }
                      JSONArray jArray =
                      (JSONArray) json.getJSONArray("results");
                      //Obtiene el valor de results del json
                      JSONObject json2 = null;

                      String [] name = null;
                      int i;

                      //Se trata el json y se obtienen los
resultados validos
                      if(jArray.length()>0){
                            json2 = jArray.getJSONObject(0);
                            name =
                      ((String)
                      json2.get("display_name")).split(",");
                      }

                      if((json2 != null) &&
     ((zoom.equals("Comunidades")) ||
  ((String) json2.get("display_name")).contains("Asturias") ||
  ((String) json2.get("display_name")).contains("Baleares") ||
  ((String) json2.get("display_name")).contains("Cantabria")||
  ((String) json2.get("display_name")).contains("Rioja")     ||
  ((String) json2.get("display_name")).contains("Navarra")) ) {
                            i = 1;
           //Si se quiere la comunidad, se coge una posicion del
     array
                      } else {
                            i = 2;
           //Si se quiere la provincia, se coge otra posicion
     del array
                      }

                      if((json2 != null) &&
                ((String)
           json2.get("display_name")).contains(";")){
                            i += 1;
                      }
                     if((json2 != null) && (name.length > i) &&
                            ((String) json2.get("display_name"))
                                        .contains("España")){
                      if((name[name.length-(i+1)].trim())
```

```
                                                    .matches("[0-9]*")){
                              location=name[name.length-
(i+2)].trim();
                        } else {

                              location=
                              name[name.length-(i+1)].trim();
                        }
                       } else {
                        location = "Extranjero";
                       }
                }


      if(!TwitterSession.counterMap.containsKey(location)){
//Si la localizacion obtenida no se encuentra en la lista
                  TwitterSession.counterMap.put(location, 1);
                  //Se crea la localización con un contador a 1
                  } else {
                        //Si se encuentra
                        int c =
                        TwitterSession.counterMap.get(location)+1;
                        //Se suma uno al contador
                  TwitterSession.counterMap.put(location, c);
                  //Se actualiza el contador de la localizacion
                  }
                  collector.emit(new Values(location,message));
                  //Se emite el resultado en el stream
      }
```

En primer lugar, este *bolt* obtiene los datos que se transfieren desde el *bolt* anterior, por lo que recibe, el mensaje con el usuario y el texto por un lado, y por otro lado la ubicación del *tuit*.

En esta etapa, solo se procesara la ubicación. El objetivo es estandarizar las ubicaciones obtenidas para que todas tengan la misma nomenclatura y poder agruparlas por regiones. Para esto, se utiliza una *url* proporcionada por *OpenStreetMap* en la cual se inserta la ubicación del *tuit* y devuelve un Json con la información de la ubicación como a que región pertenece, el país, las coordenadas, o qué tipo de territorio es.

Para ello, obtenemos la localización del *tuit*. En primer lugar se le da un formato que sea válido para la url mediante la función *encode()*. A continuación se construye la *url* y se realiza la consulta mediante otra clase creada que hemos llamado ***JsonReader.java***, la cual se muestra a continuación.

```
public String readAll(Reader rd) throws IOException{

      StringBuilder sb = new StringBuilder();
      int cp;

      while((cp = rd.read()) != -1){
            sb.append((char) cp);
      }
```

```
      return sb.toString();
}
```

Esta función, se encarga de convertir una instancia de la clase *Reader* en un String.

```
public JSONObject readJsonFromUrl(String url)
throws IOException, JSONException {

            InputStream is = new URL(url).openStream();
            //Conecta con la url externa

           try {

            BufferedReader rd =
                  new BufferedReader(
                  new InputStreamReader(is, Charset
                                              .forName("UTF-8")));
                  //Obtiene el json proporcionado por la url
            String jsonText = readAll(rd);

            //Transforma el objeto reader en un String
            String jsonTextR =
            ("{results:"+jsonText.replaceAll("#"," ")+"}");
            //Añade {results ..} a la cadena para que pueda ser
            leido como JSONObject
            JSONObject json = new JSONObject(jsonTextR);
            //Transforma la cadena en un objeto json
            return json;

           } finally {
            is.close();
           }
      }
```

Esta función realiza la consulta con la *url* y obtiene los datos proporcionados por la plataforma de *OpenStreetMap*. A continuación se procesa el Json obtenido para obtener un formato valido que pueda ser procesado en el *bolt*.

Una vez obtenido el Json como resultado de la consulta se van obteniendo los campos necesarios dependiendo de si el usuario de la aplicación ha elegido un zoom de provincias o de comunidades obteniendo un campo del array u otro.

Si el array contiene la palabra “España” se obtendrá la posición del array correspondiente a la provincia o a la comunidad autónoma, pero si el array no contiene la palabra “España”, entonces se le asignará al *tuit* la localización “Extranjero”.

Una vez obtenida la ubicación exacta de *tuit*, ésta se almacena en un *Map*. En primer lugar se comprueba si la ubicación ya existe en el Map. Si la ubicación no existe, se añade al Map como clave, junto con un contador con valor 1 como valor de esa clave. Si la ubicación ya existe en el Map, se modifica el contador que corresponde a esa ubicación incrementándolo en 1 más.

El *Map* es estático por que debe ser accesible desde otras clases del proyecto.

Una vez almacenadas todas las ubicaciones, se emiten en el *stream*, la ubicación y el mensaje del *tuit* para que sea recibido por el siguiente componente de la topología.

- **TwitterMapBolt.java**

Este es el último *bolt* de la topología, y se encarga de agrupar los *tuits* que tengan la misma ubicación.

```
public void execute(Tuple tuple){
            String location = tuple.getString(0);
            //Obtiene la localizacion de la tupla
            String message = tuple.getString(1);
            //Obtiene el mensaje de la tupla
      if((location != null) &&
                  (!location.equals("Extranjero")) &&
                              (!location.equals("Vacio"))){
            if(!TwitterSession.locationMap
                              .containsKey(location)){
            //Si no existe la localizacion, se añade al map
                        messages = new ArrayList<String>();
      //Se crea una lista asociada a la localizacion
                        messages.add(message);
      //Se añade el mensaje a la lista
            TwitterSession.locationMap.put(location, messages);
      //Se añade la clave y la lista al map
                  } else {
                        messages =
                        TwitterSession.locationMap.get(location);
                        messages.add(message);
      //Si existe la localizacion, se añade el mensaje a la lista
asociada a ella
                        TwitterSession.locationMap
                                          .put(location, messages);
                  }
            }
            collector.emit(new Values(tuple));
      //Se emite el resultado al stream
      }
```

En primer lugar, se obtienen los datos del *stream* enviados desde el *bolt* anterior, que son la localización y el mensaje del *tuit*. A su vez, el mensaje contiene el usuario y el texto del *tuit*.

Seguidamente, se comprueba la localización. Si la localización del *tuit* es “Extranjero” o “Sin ubicación”, estos no se almacenan en el *Map*.

Si por el contrario, la localización tiene un valor de una región española, en primer lugar se comprueba si el *Map* contiene esa localización entre sus claves. Si la respuesta es que no existe, entonces se crea una lista de mensajes y se le añadirá el mensaje del *tuit*. Posteriormente se le añadirá al *Map* la ubicación como clave y la lista de mensaje como valor.

Si la ubicación ya existe en el *Map*, entonces se obtiene la lista asociada a esa clave y se le añade el mensaje del *tuit*.

Al igual que en *bolt* anterior, el *Map* es estático puesto que debe ser accesible desde otras clases del proyecto.

- **TwitterMain.java**

Esta clase se encarga de construir la topología de *storm* y registrarla en el cluster para ejecutarla.

```
public void empiezaBusqueda(String f, String z, String t,
TwitterSpout tS, TwitterReaderBolt tR, TwitterWriteBolt tW,
TwitterMapBolt tM){
      f = f.trim();

      if(!(f.equals("*")) && !(z.equals("*"))){
      //Si el filtro y el zoom son distintos de '*', comienza el
      proceso

            String [] filtro = f.split("-");
            //Si el filtro contiene '-', se separan las palabras
            para ejecutar varios filtros
            Timer timer = new Timer();

            timerTask = new TimerTask() {
            //Se crea un TimerTask para ejecutar este proceso en
            otro hilo de ejecucion
                  long tiempo = Long.parseLong(t);
                        //Se obtiene el tiempo en long
                  String [] keyWords = filtro;

                  public void run() {

                        if(keyWords[0].equals("")){
                        //Si el filtro esta vacio, no se aplica
                        ningun filtro a la consulta
                              System.out.println("No hay filtros");
                              keyWords = null;
                        } else {
                              for(int i=0;i<keyWords.length;i++){
                                    System.out.println(keyWords[i]);
                              }
                        }

                        Config config = new Config();
                        config.setDebug(true);

                        TopologyBuilder builder =
                                          new TopologyBuilder();
                        //Se crea el objeto builder para construir
                  la topologia Storm
                        tS.setKeyWords(keyWords);
                        builder.setSpout("twitterSpout", tS);
                        //Se añade el spout a la topologia

                        builder.setBolt("twitterLocation", tR)
                              .shuffleGrouping("twitterSpout");
```

```
                    //Se añade el primer bolt conectado al
              spout
                    tW.setZoom(z);

        builder.setBolt("twitterMap", tW)
              .shuffleGrouping("twitterLocation");
         //Se añade el segundo bolt, conectado el primero
        bolt

        builder.setBolt("twitterMap2", tM)
              .shuffleGrouping("twitterMap");
        //Se añade el tercer bolt, conectado al segundo bolt


        LocalCluster cluster = new LocalCluster();
        //Se crea un cluster local

        cluster.submitTopology("TwitterStorm", config,
                        builder.createTopology());
        //Se registra la topologia en el cluster
              try {
                    tiempo = tiempo * 1000;
        //Se obtiene el tiempo de busqueda en milisegundos
                    synchronized(this) {
                          timerTask.wait(tiempo);
        //Se para el hilo de ejecución para evitar finalizar
la ejecucion de la topologia
                    }
              } catch (InterruptedException e1){
                    System.out.println("Fallo de ejecucion");
              }
              cluster.shutdown();
              //Se para la ejecución del cluster
              timer.cancel();
              //Se para la ejecución del timer
                  }
              };

              timer.schedule(timerTask, 0);
              //Se ejecuta el timer con un delay de 0
        }
}
```

En primer lugar, se obtienen los parámetros de la búsqueda que va a realizar el usuario que serían, las palabras por las que filtrar el *tuit*, el zoom geográfico que ha elegido, y el tiempo que desea estar filtrando *tuits*.

Si el filtro o el zoom, son iguales a "*", la ejecución no hace nada, más adelante se explicará por qué. Si el filtro y el zoom son válidos, comienza la ejecución.

Primero se comprueba si el filtro contiene "-", si es así, se dividirá el valor en palabras sueltas separándolas por el "-" especificado, lo que provocara que el filtro sea de múltiples palabras y busque los *tuits* que contengan al menos una de las palabras.

Una vez hecho esto, se crea una instancia de la clase *Timer* y otra de la clase *TimerTask*. Estas clases se utilizan para que este proceso se realice en otro hilo de ejecución en segundo plano.

A continuación, se convierte el tiempo de *String* a *long*. Seguidamente, empieza la ejecución del proceso en segundo plano. Si el filtro está vacío, se establece el array a null para que se pueda procesar en el *spout* una búsqueda por área geográfica en vez de por palabras.

Posteriormente, se crea la configuración y el objeto *builder* que se encarga de construir la topología de *storm*.

El siguiente paso es especificar los componentes de los que se compone la topología y como están conectados entre sí. El orden de los componentes será el descrito en esta sección.

La ejecución continúa con la creación de un *cluster* local en el cual se registra la topología y se ejecuta.

El tiempo se pasa a milisegundos multiplicándolo por mil y se bloquea el proceso durante este tiempo que ha especificado el usuario. Con esto se consigue que se este filtrando durante ese periodo de tiempo ya que lo ejecución no continua para desahibilitar el cluster.

Cuando el tiempo se cumple, el cluster se desabilita y se para la ejecución del *Timer* en segundo plano.

Con esto ya podremos realizar búsquedas y obtener unos datos reales acerca de los *tuits* generados en tiempo real.

8.3.2. Implementación de la interfaz

En esta sección, se especificará el funcionamiento a nivel de código fuente, de la interfaz de la aplicación.

- **Registro.jsp**

Se trata, simplemente, de un formulario de registro en el que lo interesante se encuentra en las validaciones de los campos realizados mediante *JavaScript*.

```
$(function(){
            $("#btn_registrar").click(function(){
            var campos = [""];
            var nombre = [""];
            var i;
            var ok = true;

            document.getElementById("validaReg").innerHTML="";

            for (i=1;i<6;i++){
                  (document.getElementById("registro")
```

```
                    .elements.item(i-1)).removeAttribute("class");
            //Elimina las clases de todos los campos del
            formulario
            }

            for (i=1;i<6;i++){

      nombre.push(document.getElementById("registro").elements.it
em(i-1).name); //Obtiene todos los nombres de los campos
            }

            for (i=1;i<6;i++){

      campos.push(document.getElementById("registro").elements.it
em(i-1).value); //Obtiene todos los valores de los campos
            }
            for (i=1;i<6;i++){
                  if(!((campos[i].trim()).localeCompare(""))){
//Comprueba si los campos estan vacios

      (document.getElementById("registro").elements.item(i-
1)).className = "error";   //Añade la clase error al campo vacio

      document.getElementById("validaReg").innerHTML += "El campo
"+nombre[i]+" esta vacío.<br>"; //Muestra un mensaje por cada
campo
                        ok = false
                  }
            }

            if((ok) &&
((campos[4].trim()).localeCompare(campos[5].trim()))){ //Compara
que los dos campos de contraseña sean iguales

      (document.getElementById("registro").elements.item(3)).clas
sName = "error"; //Añade la clase error a los campos contraseña

      (document.getElementById("registro").elements.item(4)).clas
sName = "error"; //si estas no coinciden
                  document.getElementById("validaReg").innerHTML
+= "Las contraseñas no coinciden. <br>"; //Muestra el mensaje de
error
                  ok = false;
            }


            return ok; //Realiza el submit del formulario
            });
      });
```

En primer lugar, se eliminan todas las clases que tienen los campos del formulario de registro. Luego se almacenan todos los nombres de los campos en un array y todos los valores de los campos en otros array.

A continuación se comprueba si los campos están vacios, y si están vacios, se le añade la clase error y se pintara por pantalla un mensaje de campo vacio.

Por último, se comprueba que los dos campos de contraseña sean iguales. Si no son iguales, se añade la clase error a los campos de contraseña y se muestra un mensaje de error.

El formulario no realizará el *submit* hasta que todas las validaciones estén cumplidas.

Una vez realizado el *submit*, los datos recogidos en el formulario se envían al servlet de registro.

- **Registro.java**

Una vez aquí, se realizan las últimas validaciones y se realiza el registro en base de datos.

```
protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

		HttpSession session = request.getSession();
		UsuarioDao uDao = new UsuarioDao();
		Usuario user = new Usuario();

		String passA = request.getParameter("Contraseña actual");
		String passN = request.getParameter("Nueva contraseña");

		if(((TwitterSession)session.getAttribute("tSession")) != null){

			if(passA !=null){
				user =
((TwitterSession)session.getAttribute("tSession")).getUsuario();

				if(passA.equals(user.getContrasena())){

					user.setContrasena(passN);
				uDao.updateUsuario(user);
	((TwitterSession)session.getAttribute("tSession")).setUsuario(user);
//Se actualiza la contraseña en sesion
					PrintWriter pw= response.getWriter();
					pw.println("<span style=\"color:
green\">Contrase&ntilde;a modificada correctamente</span>"); //Se muestra un
mensaje de confirmacion
					pw.close();

				} else {	//Si la contraseña actual no es
correcta

					PrintWriter pw= response.getWriter();
					pw.println("<span style=\"color:
red\">Contrase&ntilde;a actual incorrecta</span>"); //Se muestra un mensaje
de error
					pw.close();

				}
			} else { //Si la contraseña esta vacia
```

```
		request.getRequestDispatcher("usuario.jsp").forward(request,
response); //Se recarga la pantalla
					}
				}else { //Si la sesion no existe, el formulario es el de
registrar un usuario

					//Se obtienen los valores del formulario que ya han sido
validados
					String nombre = (String)request.getParameter("Nombre");
					String apellidos =
(String)request.getParameter("Apellidos");
					String usuario = (String)request.getParameter("Usuario");
					String pass = (String)request.getParameter("Contraseña");
					String vMensajeR = "";
					boolean okUser = true;
					String mOkRegistro = "";

					if(nombre == null){//Si el nombre es nulo

		request.getRequestDispatcher("registro.jsp").forward(request,
response); //Se redirige a registro
					} else {

							if(uDao.getUsuarioByName(usuario) != null){ //Si se
encuentra en la base de datos el usuario introducido
								vMensajeR = "El usuario ya existe";
		//El usuario estaria repetido
								okUser = false;
								request.setAttribute("vMensajeR", vMensajeR);
								request.setAttribute("okUser", okUser);

		request.getRequestDispatcher("registro.jsp").forward(request,
response); //Recarga la pagina con mensaje de error
							} else {	//Si el usuario no existe
								user.setNombre(nombre);
								user.setApellidos(apellidos);
								user.setUsuario(usuario);
								user.setContrasena(pass);
								uDao.saveUsuario(user); //Se guarda el
usuario en base de datos
								mOkRegistro = "Registro completado";
								request.setAttribute("mOkRegistro",
mOkRegistro);

		request.getRequestDispatcher("registro.jsp").forward(request,
response); //Recarga la pagina con mensaje de confirmacion
							}
					}
			}
		}
```

En esta clase, se reciben los datos del formulario de registro.

En primer lugar, se comprueba que la sesión sea nula. Si la sesión no es nula, entonces los datos del formulario son los enviados desde el formulario de cambio de contraseña.

Si la sesión es nula, entonces se está registrando un usuario nuevo. Se obtienen los datos, si el nombre es nulo, entonces es que se está intentando acceder de forma indebida

y se redireccionará al index. Si los campos están bien informados, significa que han pasado las validaciones y solo queda comprobar que el usuario no existe en base de datos. Si el usuario existe, ser redirige a la página de registro con un mensaje de error. Si no existe, se realiza el registro en base de datos y se redirige a la página de registro con un mensaje de confirmación.

- **Login.jsp**

Al igual que en el JSP de registro, solo contiene un formulario que envía los datos al servlet de Login.

- **Login.java**

En esta clase se reciben los datos enviados desde el formulario de login.

```
protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {

            HttpSession session = request.getSession();
            UsuarioDao usuarioDao = new UsuarioDao();
            BusquedaDao busquedaDao = new BusquedaDao();

            String user = request.getParameter("user");
            String pass = request.getParameter("pass");

            boolean vUser = true;
            boolean vPass = true;
            String vMensaje = "";

            //Proceso para cerrar sesion

      if((((TwitterSession)session.getAttribute("tSession")) !=
null) || (user == null) || (pass == null)){ //Si el usuario es
null y la sesion no

                  session.setAttribute("tSession", null);
      //Se establece la sesion a null

      request.getRequestDispatcher("index.jsp").forward(request,
response); //Se redirige a index

            } else {

                  if(usuarioDao.getUsuarioByName(user) == null){
//Si el usuario no existe

                        vUser = false;
                        vMensaje = "Usuario incorrecto";
                        request.setAttribute("vUser", vUser);
                        request.setAttribute("vMensaje",
vMensaje);
                        request.setAttribute("user", user);
                        session.setAttribute("tSession", null);
```

```
      request.getRequestDispatcher("Login.jsp").forward(request,
response); //Se redirige a index con un mensaje de error

                  }else{


      if(!((usuarioDao.getUsuarioByName(user).getContrasena()).eq
uals(pass))){ //Si la contraseña no es correcta

                  vPass = false;
                  vMensaje = "Contraseña incorrecta";
                  request.setAttribute("vPass", vPass);
                  request.setAttribute("vMensaje", vMensaje);
                  request.setAttribute("user", user);
                  session.setAttribute("tSession", null);

                  request.getRequestDispatcher("Login.jsp")
                                    .forward(request, response);
                  //Se redirige a index con un mensaje de error

      } else {    //Si usuario y contraseña son correctos

            TwitterSession twitterSession = new TwitterSession();

            Usuario usuario = usuarioDao.getUsuarioByName(user);

            twitterSession.setUsuario(usuario);
            //Se establece el usuario en sesion
            twitterSession.setBusquedas(
                  busquedaDao.getBusquedasByUsuario(usuario));
            //Cargan en sesion las busquedas del usuario

            session.setAttribute("tSession", twitterSession);
            //Se establece la sesion

            request.getRequestDispatcher("index.jsp")
                        .forward(request, response);
            //Se redirige a index

                        }
                  }
            }
      }
```

En primer lugar, se comprueba que la sesión sea nula. Si la sesión no es nula, es que se ha pulsado en cerrar sesión.

A continuación, se comprueba las credenciales introducidas en el formulario. Primero se comprueba si el usuario existe, si no existe, se redirecciona a login con un mensaje de error. Si el usuario existe, se comprueba si a contraseña almacenada en base de datos coincide con la introducida. Si coinciden se redirecciona a index y si no, se redirecciona a login con un mensaje de error.

- **Index.jsp**

Esta JSP es la principal de la aplicación, aquí aparece un formulario el cual rellena el usuario para establecer los parámetros de una búsqueda. Como en las demás JSP, solo se pinta el formulario y lo más interesante es el *JavaScript* que se lanza al enviar el formulario.

```
$(function(){
      $("#btn_enviar").click(function(){

            var url = "mapa.jsp";
            //jsp del que se obtiene la respuesta
             var filtro = document.getElementById("formulario")
                                  .elements.item(0).value;
            //Obtiene el valor del campo numero 1 del formulario
            var zoom = document.getElementById("formulario")
                                  .elements.item(1).value;
            //Obtiene el valor del campo numero 2 del formulario
            var time = document.getElementById("formulario")
                                       .elements.item(2).value;
            //Obtiene el valor del campo numero 3 del formulario


            document.getElementById("respuesta").innerHTML = "";
                  //Pone en blanco el div con ese id

            document.getElementById("validacion").innerHTML = "";
            document.getElementById("final").innerHTML = "";

            if(time < 30){

                  document.getElementById("validacion")
                        .innerHTML = "El tiempo es incorrecto";
                  //Pinta mensaje de error en el div con ese id
                   $('#iTiempo').addClass("error");
                  //Añade la clase 'error' al div con ese id

            } else {

                    $('#iTiempo').removeClass("error");
                    $('#carga').addClass("cargaVisible");
                    $('#btn_enviar').addClass("noEnviar");
                    $('#btn_parar').removeClass("noParar");

                    $.ajax({
                       type: "POST",
                       url: url,
                       data: $("#formulario").serialize(),
             // Adjuntar los campos del formulario enviado.
                        beforeSend: function() {

                              $('#carga').addClass("cargaVisible");
                        },
                        success: function(data){
```

```
                        $("#respuesta").html(data);
                    // Mostrar la respuestas obtenidas del jsp
                    y pintarlas en el div con id 'respuesta'.
                     time = (time * 1000)+5000;
                     refrescar = setInterval("refresh()",5000);
                    //Ejecutar la funcion 'refresh()' cada 5
                seg

setTimeout("cargando()",6000);
//Ejecutar la funcion 'cargando()' cuando pasen 6 segundos
 stopTimer = setTimeout("stopFunction()",time);
//Ejecutar la funcion 'stopFunction()' cuando pase el tiempo
noEnviarTimer = setTimeout("noEnviar()",time);
//Ejecutar la funcion 'noEnviar()' cuando pase el tiempo
            }
      });
}
return false; // Evitar ejecutar el submit del formulario.
      });
});
```

Al pulsar el botón enviar, se pone en marcha el script, que valida el tiempo introducido, y envía los datos a otro JSP sin que se realice el *submit* del formulario. Con esto se consigue que no se cambie de página en la interfaz. Los resultados obtenidos de la otra JSP se muestran en la misma página de index, empotrándola en un div. La JSP que envía los resultados sería la representación de los *tuits* en el mapa y se recargaría cada 5 segundos con la función *refresh()*.

```
function refresh(){
      $("#respuesta").load("mapa.jsp"); //Recarga el jsp
      return false;
}
```

La recarga se va produciendo durante el tiempo de búsqueda que ha especificado el usuario. Cuando este tiempo acaba, se anula la recarga de la página.

- **Mapa.jsp**

Esta JSP se encarga lanzar la búsqueda con los datos obtenidos del formulario de index y de pintar los mapas y colorear las regiones según los tuits que se obtengan.

Los mapas se pintan mediante SVG indicando las coordenadas de los puntos que, uniéndolos, forman la silueta de una región concreta.

Al pulsar sobre una región, se mostrará el contenido del *Map* estático que se explicó en apartados anteriores.

Este JSP se recargará cada 5 segundos debido a la ejecución del div anterior por lo que la primera vez obtendrá los campos informados y podrá lanzar la búsqueda, pero después de ésta ejecución, los campos vienen nulos y se producirían fallos ya que intentaría empezar la búsqueda con estos campos nulos. Para solucionarlo, se establece que si los campos vienen nulos, se cambian su valor a “*” que es la explicación que tiene la condición del código de la clase TwitterMain.java.

Con esto concluye un resumen de las principales características de la aplicación, obviamente no está todo el código.

El código al completo junto al *JavaDoc* generado por *Eclipse*, se adjuntan en un cd junto a esta documentación.

9. Pruebas

Para garantizar el correcto funcionamiento de la aplicación, es necesario realizar las pruebas pertinentes para detectar cualquier fallo que pueda alterar la rutina de ejecución del proyecto.

Para ello, se han realizado distintos tipos de prueba que se detallarán en este capítulo.

9.1. Pruebas unitarias

Consisten en realizar pruebas individuales de cada funcionalidad de la aplicación de forma aislada para comprobar su funcionamiento por separado.

9.1.1. Pruebas durante el desarrollo

Las primeras pruebas que se realizan al proyecto, tienen lugar durante el desarrollo del mismo. Así, se han realizado pequeñas pruebas de funcionamiento al término del desarrollo de cada funcionalidad con las que cuenta la aplicación.

- La primera de las pruebas, se realiza para verificar que ***Storm*** **conecta** correctamente con **Twitter**, y es capaz de **recibir datos** por parte de la red social. Estas pruebas están relacionadas directamente con el *spout* de la topología *Storm*.

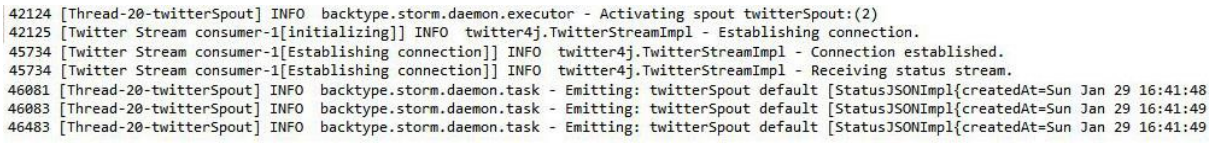


Ilustración 23: Prueba conexión Storm - Twitter

- La siguiente prueba que se realizó, fue el **obtener y almacenar** las diferentes **localizaciones** registradas en la búsqueda, con su nombre exacto y la cantidad de *tuits* que se han registrado en cada una de ellas. Esta prueba está directamente relacionada con la ejecución de los *bolts* de la topología *Storm*.

```
Result: La Rioja : 1
Result: Extranjero : 20
Result: País Vasco : 4
Result: Cataluña : 11
Result: Galicia : 4
Result: Comunidad de Madrid : 9
Result: Asturias : 2
Result: Aragón : 3
Result: Castilla-La Mancha : 2
Result: Islas Baleares : 1
Result: Canarias : 2
Result: Castilla y León : 6
Result: Murcia : 2
Result: Cantabria : 2
Result: Andalucía : 11
Result: Extremadura : 2
Result: Comunidad Valenciana : 5
```

Ilustración 24: Prueba lista localizaciones

- Por último, en las pruebas relacionadas con la topología de *Storm*, se realizó una prueba para ver si se obtenía de forma adecuada el formato de usuario y mensaje de cada *tuit* a la vez que se relacionaba el *tuit* con su ubicación. Esta prueba, como la anterior, también está directamente relacionada con la ejecución de los *bolts* de la topología *Storm.* En la siguiente *ilustración 25* se puede ver cómo se obtienen los *tuits* con su ubicación, y se muestra el usuario en forma de enlace que redirige a la cuenta de usuario junto con el mensaje que contiene el *tuit*. Como se puede observar, el número de *tuits* con su ubicación que se muestran en la *ilustración 24* coincide con el número de *tuits* que se han registrado en una ubicación de la *ilustración 24*.

```
Mensaje: [La Rioja] <a href="https://twitter.com/CDAnguiano_" target="_blank">@CDAnguiano_</a> dijo: Descanso en isla @CDAnguiano_ 1 Sotes
Mensaje: [País Vasco] <a href="https://twitter.com/lady__cherry" target="_blank">@lady__cherry</a> dijo: ?Norteñas máximas? es que te quier

No sin… https://t.co/i6aWvVLcl4
Mensaje: [País Vasco] <a href="https://twitter.com/RelojBilbao" target="_blank">@RelojBilbao</a> dijo: ?
Pan Pan Pan Pan Pan
Mensaje: [País Vasco] <a href="https://twitter.com/VG_Policia" target="_blank">@VG_Policia</a> dijo: Consejos para que el frío no impida la
Mensaje: [País Vasco] <a href="https://twitter.com/manolo_nolin" target="_blank">@manolo_nolin</a> dijo: Muy buenas olas en Bakio. Domingo
Mensaje: [Cataluña] <a href="https://twitter.com/PauCostaRibes20" target="_blank">@PauCostaRibes20</a> dijo: M'encanta aquest mapa d'Europa
PD: I falta Catalunya!!!
#Europa #ContinentEuropeu https://t.co/46quBTrlYu
Mensaje: [Cataluña] <a href="https://twitter.com/RelojHospitalet" target="_blank">@RelojHospitalet</a> dijo: ?
Tintán Tintán Tintán Tintán Tintán
Mensaje: [Cataluña] <a href="https://twitter.com/hoodnightss" target="_blank">@hoodnightss</a> dijo: acabo de venir de una comida familiar
Mensaje: [Cataluña] <a href="https://twitter.com/mireiagusti" target="_blank">@mireiagusti</a> dijo: #N1CanalFiesta4 Voto por "Mi Vecina" d
Mensaje: [Cataluña] <a href="https://twitter.com/Wicked_Wave" target="_blank">@Wicked_Wave</a> dijo: En serio super hermosa y sep pareces u
Mensaje: [Cataluña] <a href="https://twitter.com/MeteoAlmoster" target="_blank">@MeteoAlmoster</a> dijo: #Almoster, #BaixCamp, 16:00 UTC -
Mensaje: [Cataluña] <a href="https://twitter.com/hadgigi" target="_blank">@hadgigi</a> dijo: i was gonna mention m***** but they keep stalk
Mensaje: [Cataluña] <a href="https://twitter.com/albertpave" target="_blank">@albertpave</a> dijo: Petition: Prevent Donald Trump from maki
Mensaje: [Cataluña] <a href="https://twitter.com/RelojBarcelona" target="_blank">@RelojBarcelona</a> dijo: ?
Bang Bang Bang Bang Bang
Mensaje: [Cataluña] <a href="https://twitter.com/atrujillo90" target="_blank">@atrujillo90</a> dijo: 16:52 Temp. 13.6°C, Hum. 58%, Dewp. 4.
Mensaje: [Cataluña] <a href="https://twitter.com/Etnosistema" target="_blank">@Etnosistema</a> dijo: @badocheroquee Cobrar més de 1000 eurc
Mensaje: [Galicia] <a href="https://twitter.com/meteolouro1" target="_blank">@meteolouro1</a> dijo: Wind 2,2 km/h SSE. Barometer 1008,61 hP
Mensaje: [Galicia] <a href="https://twitter.com/RelojMariaPita" target="_blank">@RelojMariaPita</a> dijo: ?
Tintón Tintón Tintón Tintón Tintón
Mensaje: [Galicia] <a href="https://twitter.com/RelojVigo" target="_blank">@RelojVigo</a> dijo: ?
Clan Clan Clan Clan Clan
Mensaje: [Galicia] <a href="https://twitter.com/MeteoOrense" target="_blank">@MeteoOrense</a> dijo: 16:48 Temp. 15.7°C, Hum. 77%, Dewp. 10.
```

Ilustración 25: Prueba tuits

Una vez realizadas estas pruebas con éxito, se podía continuar con el desarrollo, pues la esencia de la aplicación funcionaba correctamente.

9.1.2. Pruebas al finalizar el desarrollo

Al finalizar el desarrollo, se realizaron todas las pruebas que tienen que ver más con la interfaz y el uso que le puede dar el usuario a la aplicación. Estas pruebas consisten en comprobar el funcionamiento de las validaciones de los formularios, el acceso a base de datos de la aplicación y el acceso del usuario a las diferentes interfaces de la aplicación.

- El acceso a base de datos está controlado por hibernate y por las clases *DAO* implementadas que se han explicado en apartados anteriores. Esta prueba solo consiste en comprobar que hibernate crea la base de datos si esta no existe, y que puede almacenar o gestionar los datos que se almacenan en su interior una vez creada.

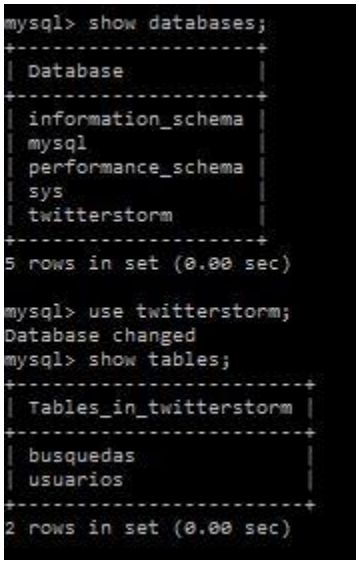


Ilustración 26: Prueba de creación base de datos

Ilustración 27: Prueba de conexión con hibernate

- Una vez que se ha realizado la comprobación de que hay conexión con base de datos, se comprueba que un usuario que no tenga una sesión iniciada, no podrá acceder a ninguna interfaz que no sea el *login*, incluso introduciendo la url de la interfaz en el buscador.

- Por último, se comprueba que las validaciones de los diferentes formularios funcionan correctamente.

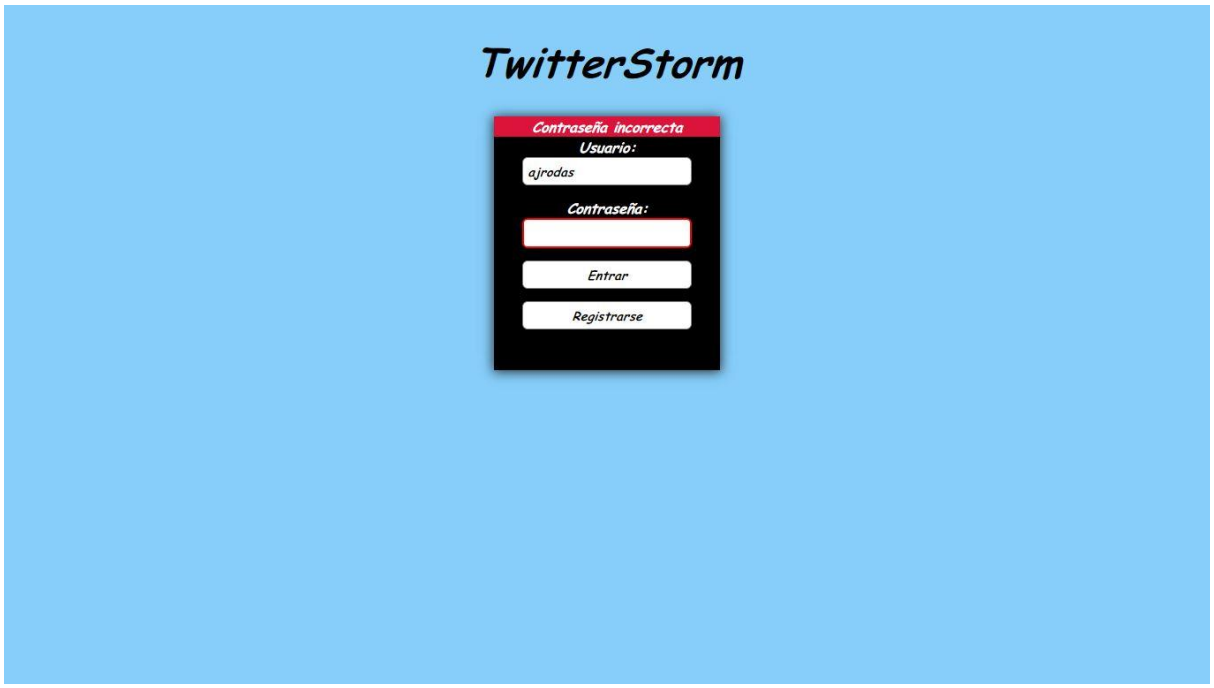


Ilustración 28: Prueba validación contraseña Login

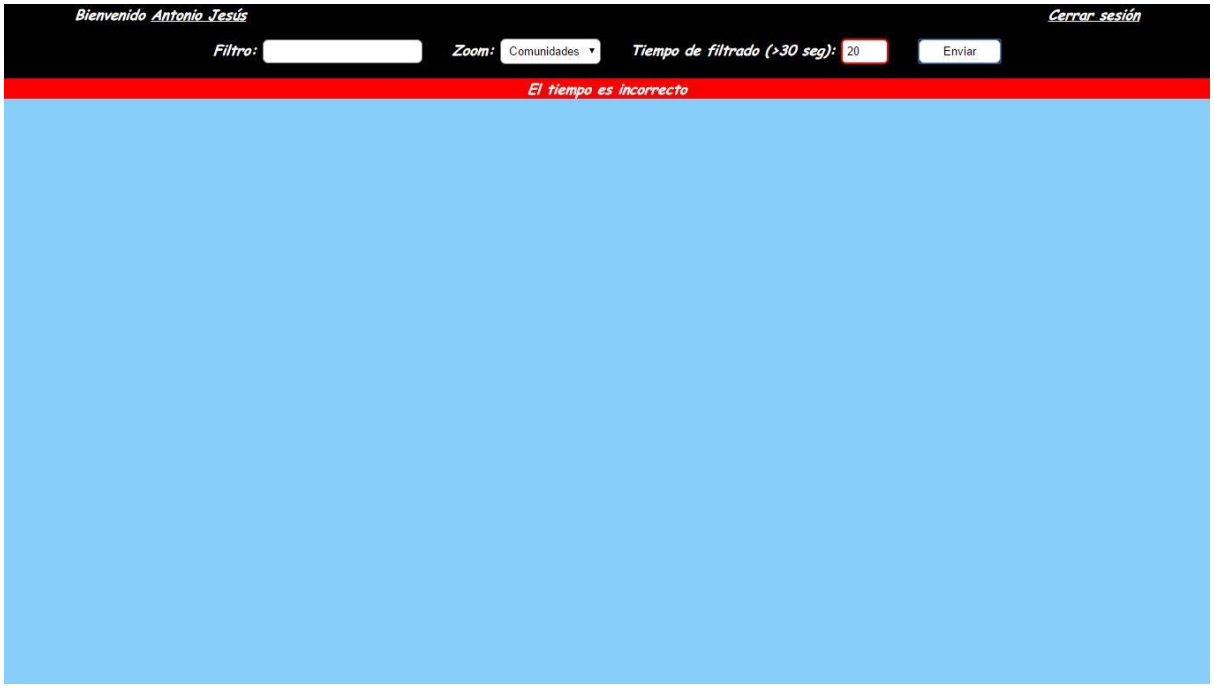


Ilustración 29: Prueba validación tiempo de búsqueda

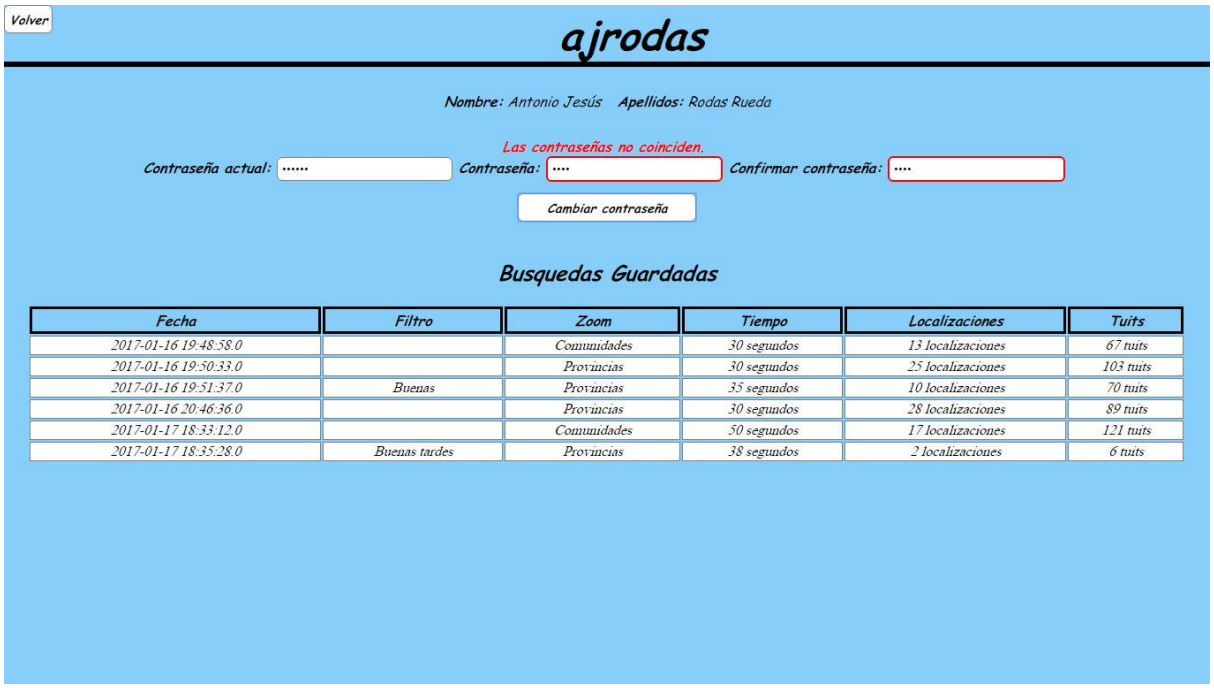


Ilustración 30: Prueba validación cambio contraseña

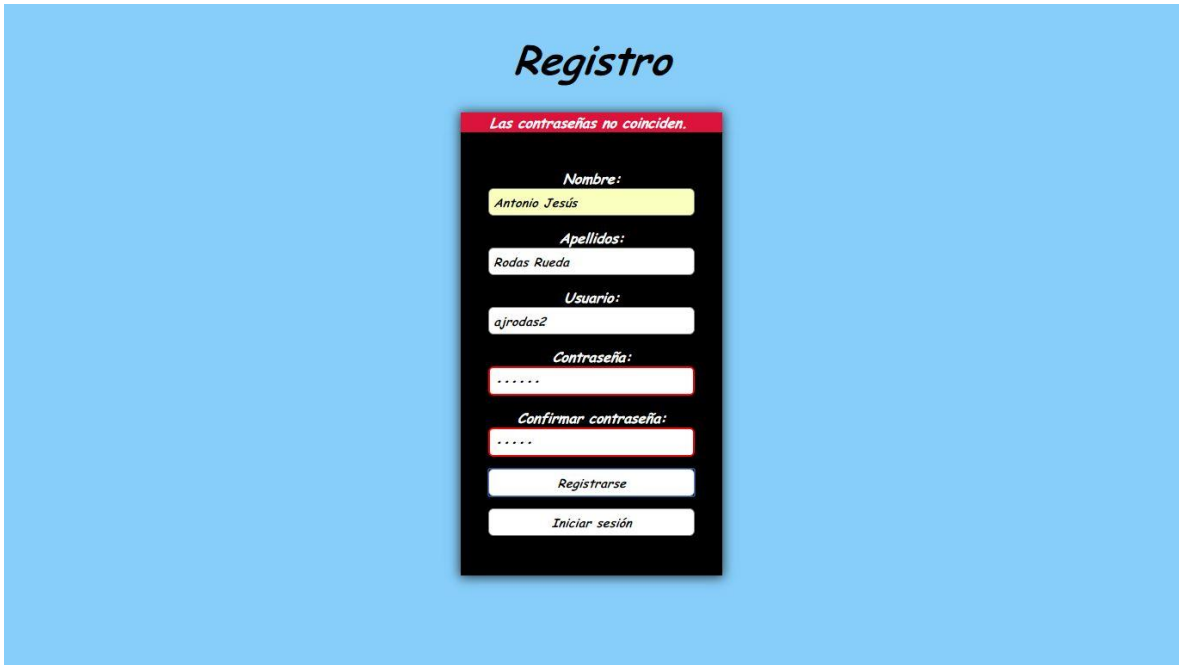


Ilustración 31: Prueba validación contraseña registro

9.2. Pruebas de integración

Para concluir, se comprueba el funcionamiento de todos los componentes que forman la aplicación funcionando en conjunto. Para esto, se realiza una iteración completa de la aplicación, que consiste en realizar un registro, iniciar sesión, realizar una búsqueda, guardar la búsqueda, ver el perfil de usuario y cerrar la sesión.

Una vez completado el proceso, se puede asegurar que la aplicación tiene un correcto funcionamiento.

10. Conclusiones y trabajo futuro

Como conclusión, me gustaría especificar tres aspectos respecto al proyecto que creo que son importantes que se describan y que son consecuencia directa del trabajo realizado durante todos estos meses.

10.1. Conocimientos adquiridos

Los conocimientos adquiridos durante el desarrollo del proyecto son de un valor incalculable a mi parecer. Se obtenido conocimientos acerca del área informática por supuesto, pero por otra parte querría resaltar los conocimientos aprendidos que no tienen tanta relación con la materia como son la organización del tiempo de trabajo; realizar investigaciones sobre tecnologías que no se han aprendido anteriormente, lo que conlleva que se hayan obtenido ciertas capacidades a la hora de saber buscar información acerca de un tema, dónde hacerlo y cómo hacerlo; indirectamente, se ha reforzado el nivel de inglés, ya que casi toda la documentación encontrada para el proyecto, o más bien toda, se encontraba en este idioma; y además, se ha aprendido a redactar una documentación correctamente con una apariencia elegante y legible.

Por otra parte, en el tema de la informática, se han obtenido conocimientos, principalmente, de las capacidades de los lenguajes de programación usados para el desarrollo como son *Java, HTML, CSS, JavaScript o Ajax*. Por supuesto se ha aprendido bastante acerca del funcionamiento tanto de *Apache Storm* como de Twitter. Además, se han conocido herramientas que no se conocían o no se sabían usar, como son *Hibernate* o el software de control de versiones *Git*.

10.2. Futuro del proyecto

En mi opinión personal, creo que el proyecto tiene muchísimo trabajo futuro, principalmente porque en mi caso no he visto ninguna otra aplicación que haga lo que es capaz de hacer la aplicación que se ha desarrollado en este proyecto.

Entre otras cosas, el trabajo futuro está marcado por las posibilidades que ofrece *Storm*, que obviamente, con los recursos que se han tenido disponibles durante el desarrollo, no se han explotado al máximo.

Por otra parte creo que la esencia de la aplicación, está desarrollada y explicada, y que los posibles cambios serían cambios para añadir funcionalidades o aumentar el rendimiento. Algunos ejemplos de estos cambios podrían ser el agrandar el radio geográfico y en lugar de trabajar sobre los *tuits* que se reciben en España, hacerlo sobre los *tuits* de todo el mundo, o incluso ofrecer más posibilidades en el tema de manejo de la información almacenada en base de datos.

En resumen, creo que el trabajo futuro sería ampliar la características y posibilidades del proyecto, pero manteniendo su esencia o su característica principal como es el filtrar los *tuit* y situarlos en un mapa.

10.3. Valoración personal

Como valoración personal, tengo que destacar que siento que el proceso de desarrollo de este proyecto me ha ayudado a darme cuenta lo que de verdad me gusta de la informática y que no me equivoqué a la hora de escoger esta profesión.

En mi caso, antes de empezar con el TFG no tenía muy claro el área de la informática a la que me quería dedicar de lleno, pero sin pensarlo, me he dado cuenta de que me he ido orientando poco a poco al tema de desarrollo y maquetado de aplicaciones web y la programación *Java*.

Al finalizar el TFG, he sentido un poco de "rabia" ya que una vez que he realizado todo el trabajo de investigación, que es lo que más tiempo me ha llevado, es cuando se me han ocurrido más funcionalidades para añadir a la aplicación.

En conclusión, estoy bastante satisfecho del trabajo realizado.

10.4. Conclusiones

En este TFG, se ha desarrollado un sistema de procesamiento de una gran cantidad de información generada en tiempo real mediante el uso de tecnologías punteras como *Apache Storm* y recursos para obtener información que se usan actualmente como son las redes sociales, en concreto Twitter.

Además se ha implementado un proceso de obtención de información totalmente parametrizable cuyo fin es obtener una información completamente personalizada para cada usuario.

El sistema es capaz de:

- Capturar los *tuits* que se producen en Twitter en tiempo real.
- Realizar la captura de *tuits* durante un periodo determinado de tiempo que es especificado por el usuario.
- Parametrizar la búsqueda mediante una palabra o un conjunto de palabras, obteniendo solo los *tuits* que contengan esos términos especificados por el usuario.
- Representar gráficamente los *tuits* obtenidos en un mapa en función de la localización desde la que se produzca cada uno de ellos.

- Establecer el nivel de detalle geográfico en el que se quieren representar los *tuits*, siendo posible representarlos en regiones divididas en provincias o en comunidades autónomas de España.

Así pues, este sistema permite que los usuarios finales, puedan visualizar la información que se está produciendo en las redes sociales en tiempo real sin que sea necesario que posean conocimiento alguno en las tecnologías usadas para su implementación.

Bibliografía

[1] Estadísticas de twitter https://www.brandwatch.com/es/2016/06/44-estadisticas-twitter-2016/
[Fecha de último acceso: 30 de Enero de 2017]

[2] BoundinBox Coordenadas http://boundingbox.klokantech.com/
[Fecha de último acceso: 30 de Enero de 2017]

[3] Tendencias Twitter https://365.weflive.com/#!/r/worldwide/t/all-topics
[Fecha de último acceso: 30 de Enero de 2017]

[4] Nathan Marz http://nathanmarz.com/
[Fecha de último acceso: 30 de Enero de 2017]

[5] Twitter Analytics https://analytics.twitter.com/about
[Fecha de último acceso: 30 de Enero de 2017]

[6] Hadoop http://hadoop.apache.org/
[Fecha de último acceso: 30 de Enero de 2017]

[7] Stateless https://es.wikipedia.org/wiki/Protocolo_sin_estado
[Fecha de último acceso: 30 de Enero de 2017]

[8] Twitter Developers https://dev.twitter.com/docs
[Fecha de último acceso: 30 de Enero de 2017]

[9] Estadísticas redes sociales https://www.brandwatch.com/es/2016/08/96-estadisticas-redes-sociales-2016/
[Fecha de último acceso: 30 de Enero de 2017]

[10] Open Street Map https://www.openstreetmap.org/
[Fecha de último acceso: 30 de Enero de 2017]

[11] DIA https://wiki.gnome.org/Apps/Dia
[Fecha de último acceso: 30 de Enero de 2017]

[12] GanttProyect http://www.ganttproject.biz/
[Fecha de último acceso: 30 de Enero de 2017]

[13] Apps Twitter https://apps.twitter.com/
[Fecha de último acceso: 30 de Enero de 2017]

[14] Clojure https://clojure.org/
[Fecha de último acceso: 30 de Enero de 2017]

A. Manual de instalación

La instalación de la aplicación, en este caso, consiste en desplegar la aplicación en un servidor web. Para eso tendremos que crear un archivo *war* desde eclipse de la siguiente forma:

- Ejecutamos *Eclipse*, y en la pantalla principal, seleccionamos, arriba a la izquierda la opción **File > Export…**

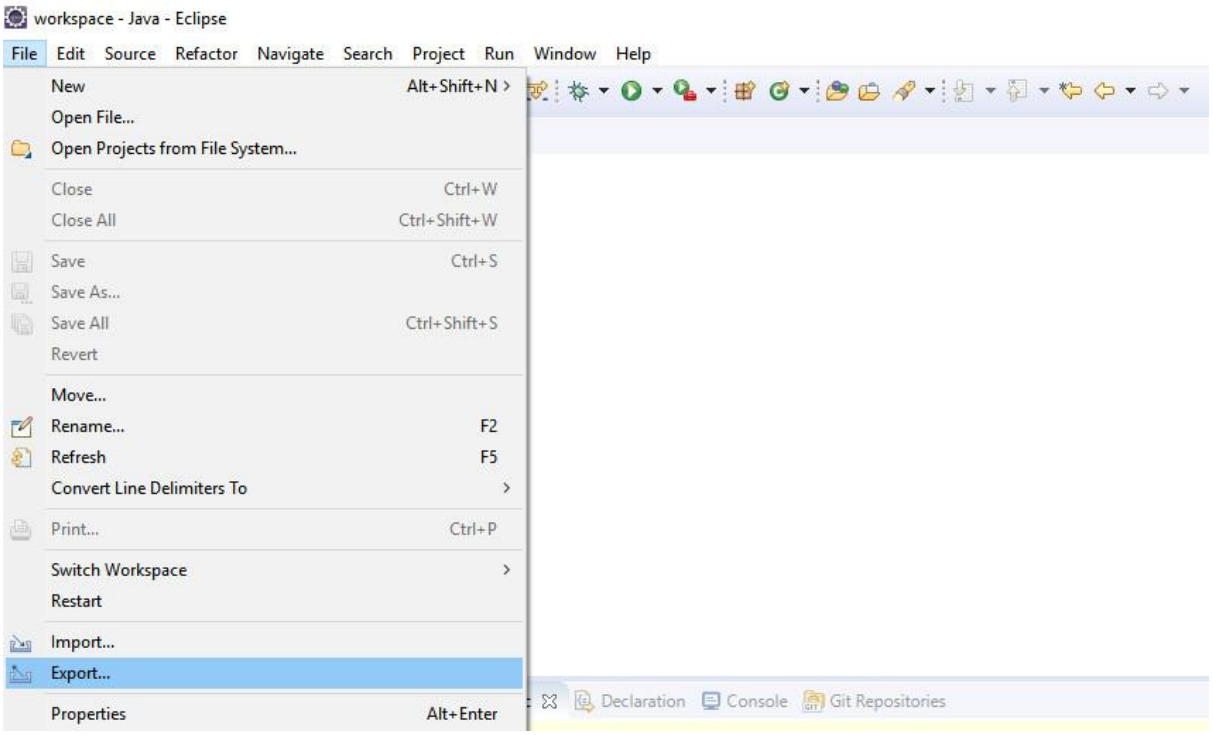


Ilustración 32: Exportar proyecto

- A continuación, ponemos en el buscador la palabra "*war*", seleccionamos **"WAR file"** y pulsamos en siguiente.

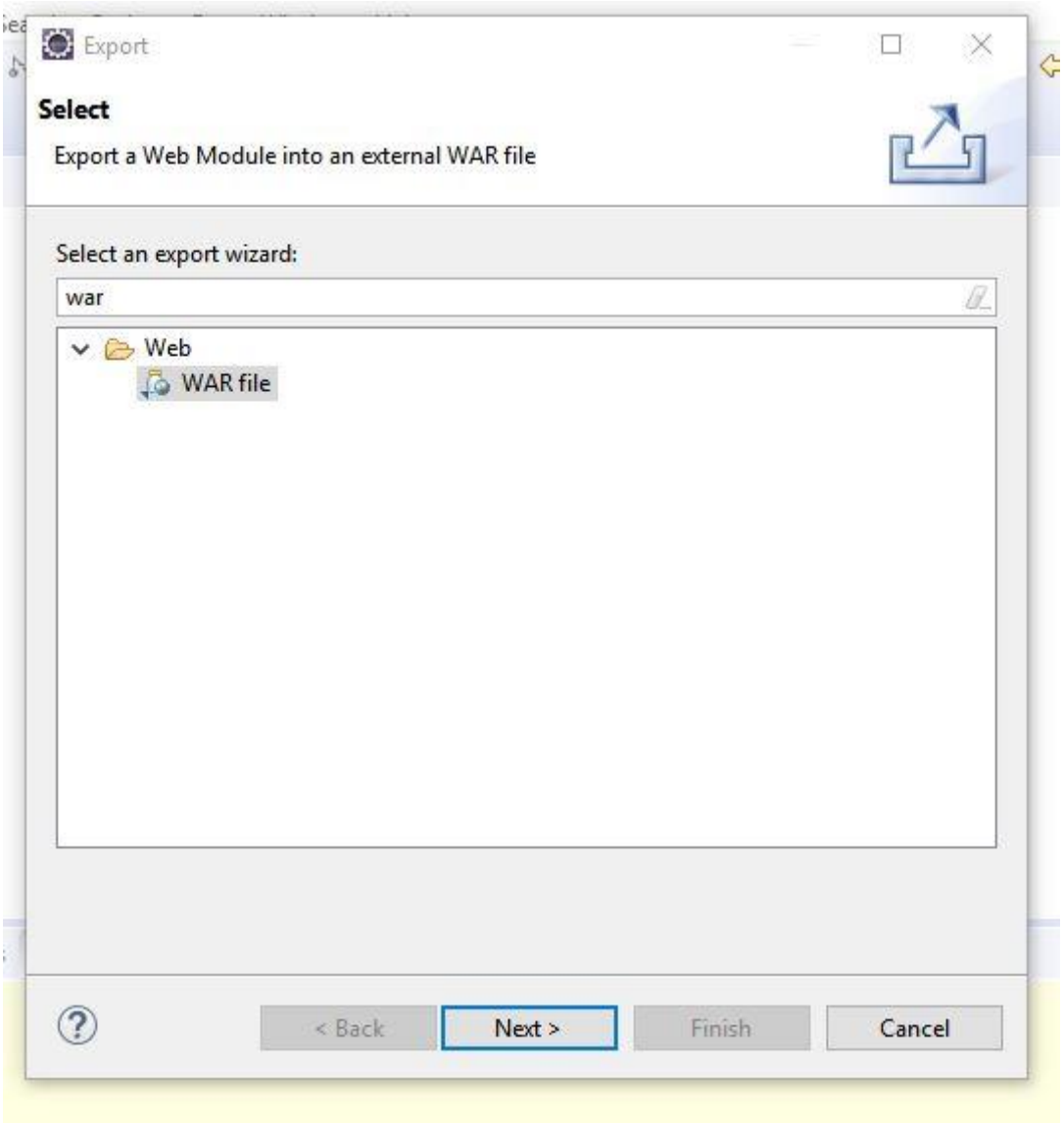


Ilustración 33: Archivo war

- Por último, seleccionamos el proyecto web que queremos exportar a *war*, en este caso *"TwitterStorm",* y el directorio de destino donde vamos a guardar el archivo. Pulsamos en finalizar.

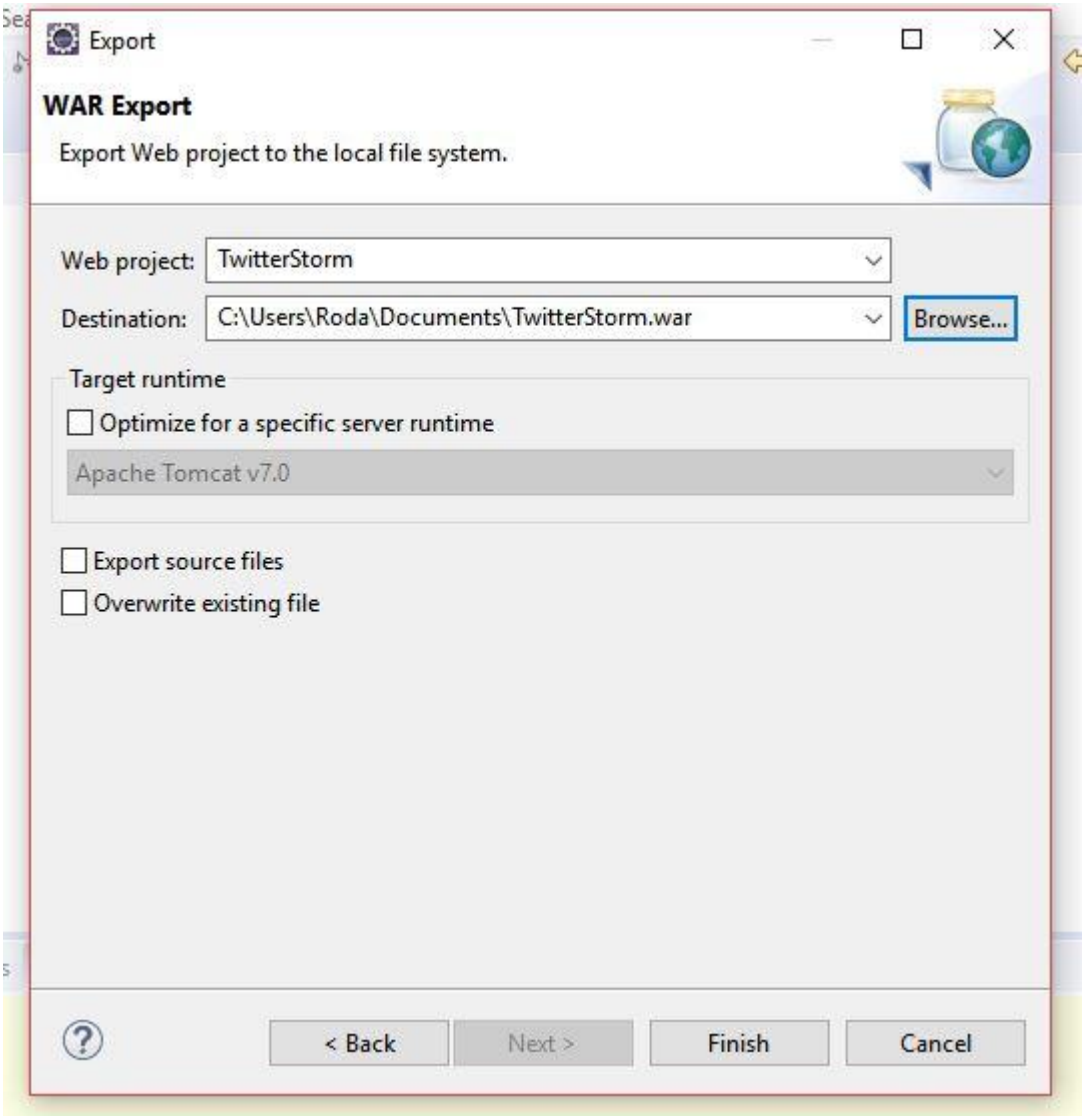
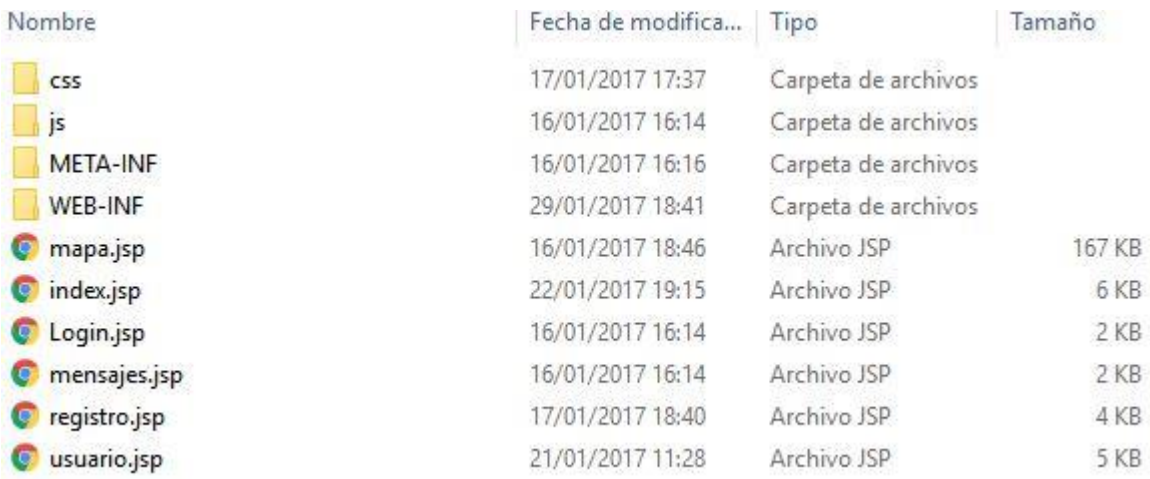


Ilustración 34: Exportar a war

- Este proceso generará un archivo que puede ser abierto con el programa *winrar* para comprobar que se haya generado completamente. La estructura final del archivo debe ser así:



Nombre	Fecha de modifica...	Tipo	Tamaño
css	17/01/2017 17:37	Carpeta de archivos	
js	16/01/2017 16:14	Carpeta de archivos	
META-INF	16/01/2017 16:16	Carpeta de archivos	
WEB-INF	29/01/2017 18:41	Carpeta de archivos	
mapa.jsp	16/01/2017 18:46	Archivo JSP	167 KB
index.jsp	22/01/2017 19:15	Archivo JSP	6 KB
Login.jsp	16/01/2017 16:14	Archivo JSP	2 KB
mensajes.jsp	16/01/2017 16:14	Archivo JSP	2 KB
registro.jsp	17/01/2017 18:40	Archivo JSP	4 KB
usuario.jsp	21/01/2017 11:28	Archivo JSP	5 KB

Ilustración 35: Referencia war

Una vez finalizado el proceso, solo habrá que desplegar el archivo en la carpeta del servidor correspondiente.

B. Manual de usuario

A continuación se muestra un sencillo tutorial de cómo sería el uso de la aplicación para un usuario.

En primer lugar, deberemos registrarnos en la aplicación, ya que se nos pide usuario y contraseña para acceder.

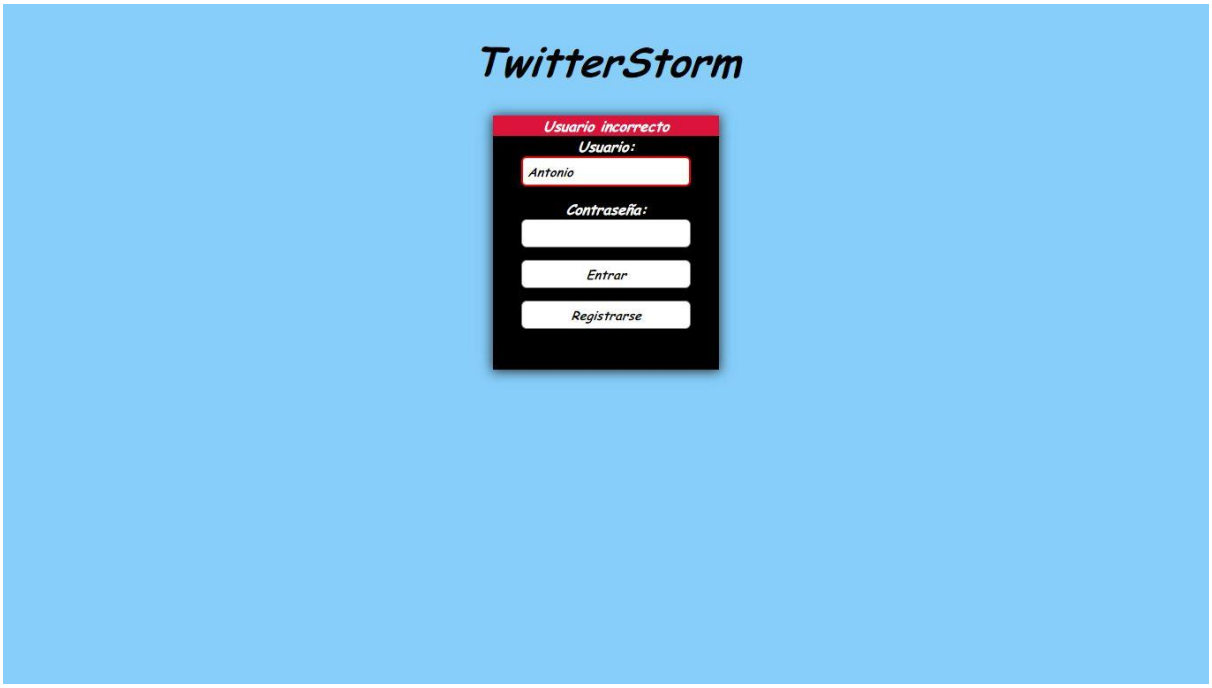


Ilustración 36: Usuario incorrecto Login

Para ello, pinchamos en el botón “Registrarse”.

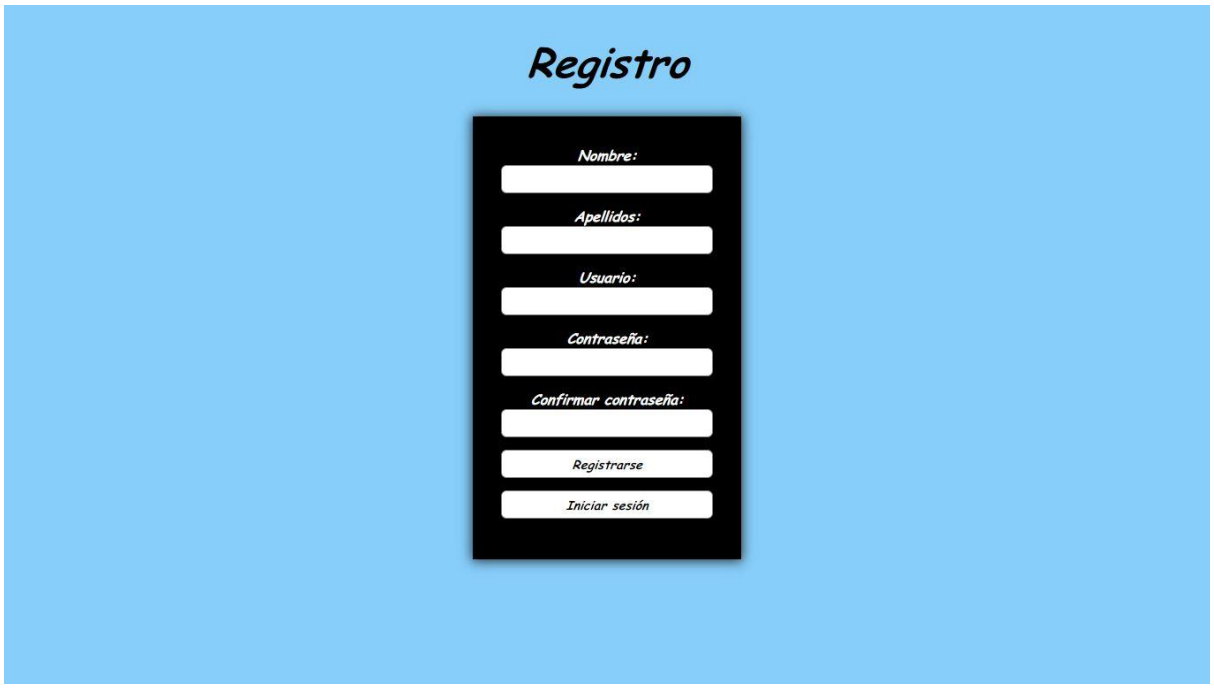


Ilustración 37: Registro

Una vez llegados a esta pantalla, rellenamos los datos correspondientes teniendo en cuenta que el usuario debe ser un valor que no esté asignado a otro usuario.

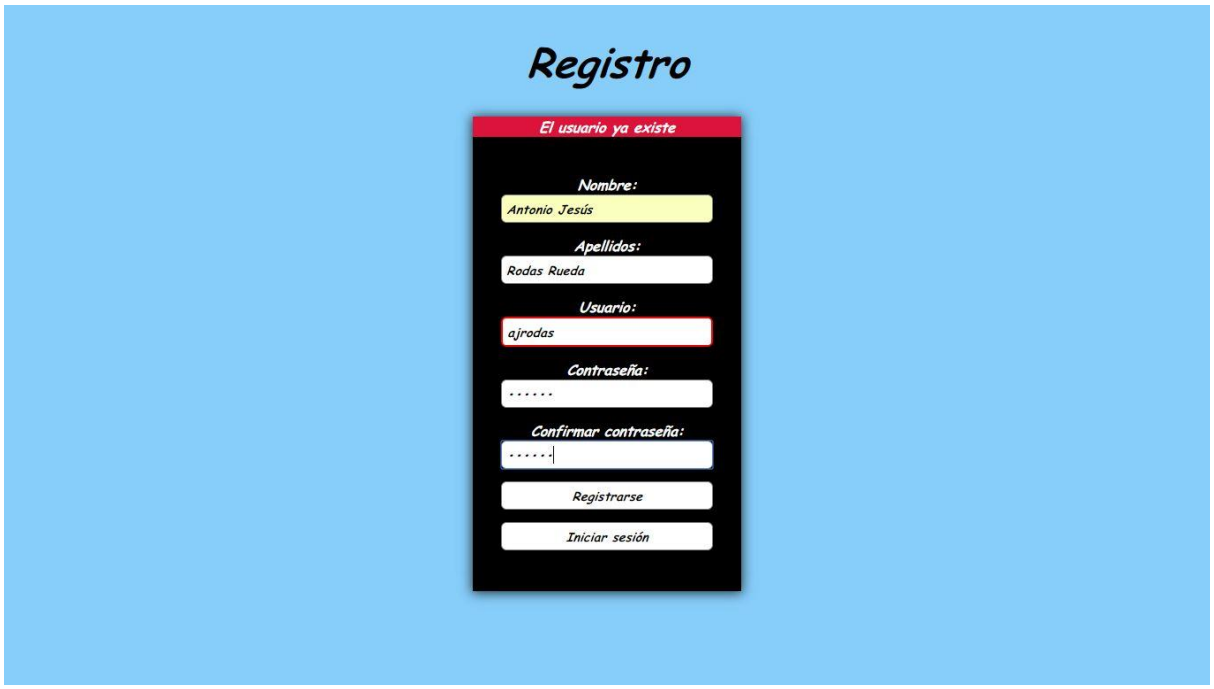


Ilustración 38: Usuario existente Registro

Una vez realizado el registro, pinchamos en el botón “Iniciar sesión” y podremos iniciar sesión en la aplicación con el usuario y la contraseña que hemos especificado en el registro.

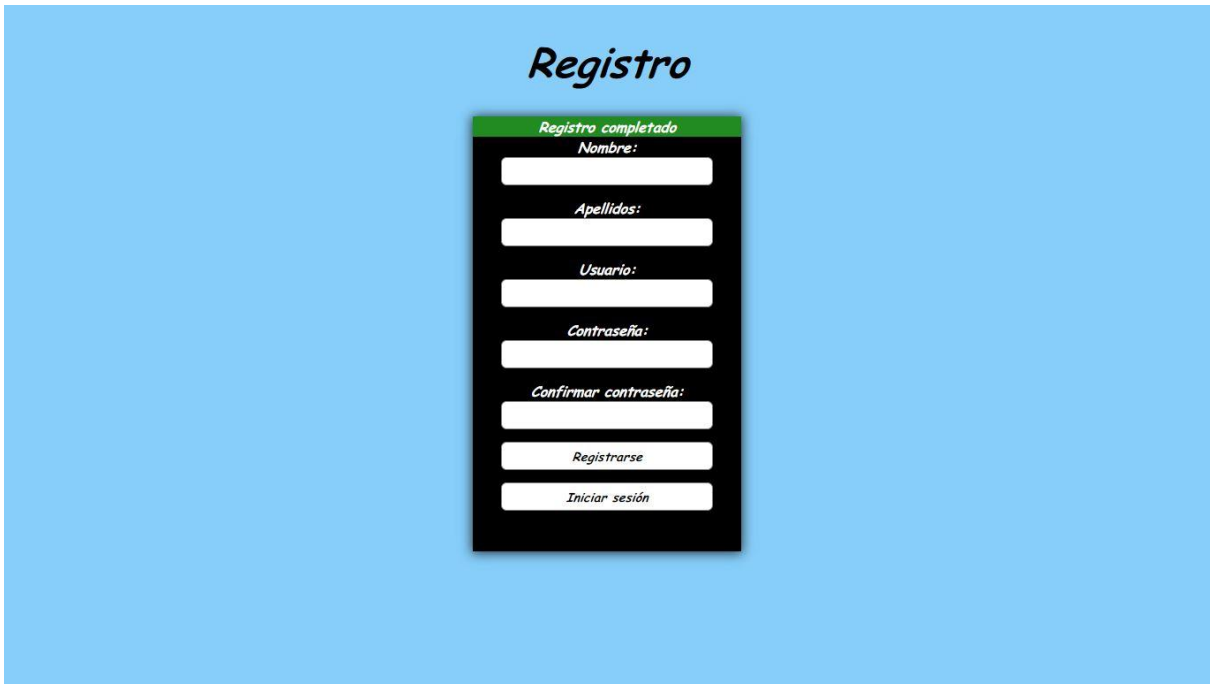


Ilustración 39: Registro completado

Cuando iniciamos sesión, aparece una pantalla en la cual se permite introducir los parámetros que se van a establecer en la búsqueda que se va a realizar.

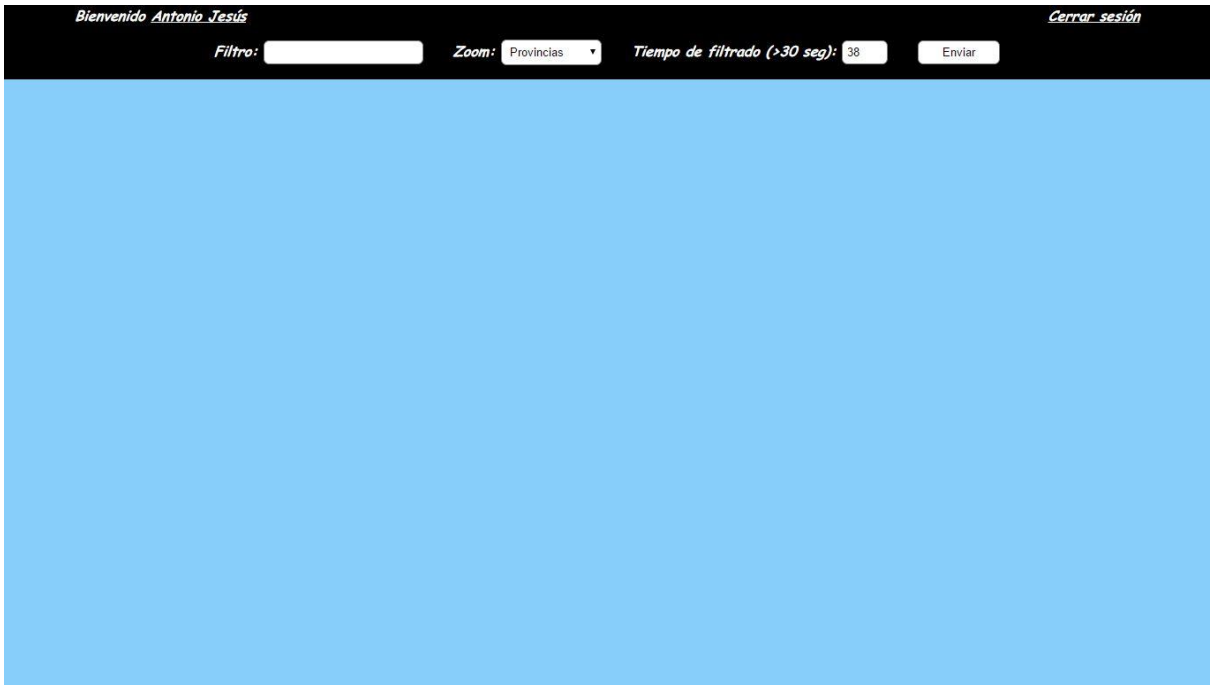


Ilustración 40: Inicio

Una vez que se hayan introducido los parámetros deseados, pinchamos en "Enviar" y la aplicación empezará a filtrar los *tuits* que se vayan produciendo que cumplan los

requisitos que se han especificado en la búsqueda y empezará a colorear las distintas regiones del mapa conforme se vayan encontrando los *tuits* en esa ubicación.

Ilustración 41: Iniciar búsqueda

La búsqueda terminará cuando el tiempo especificado se acabe o se pulse en el botón "Parar" y se indicará mediante un mensaje que la búsqueda ha terminado.

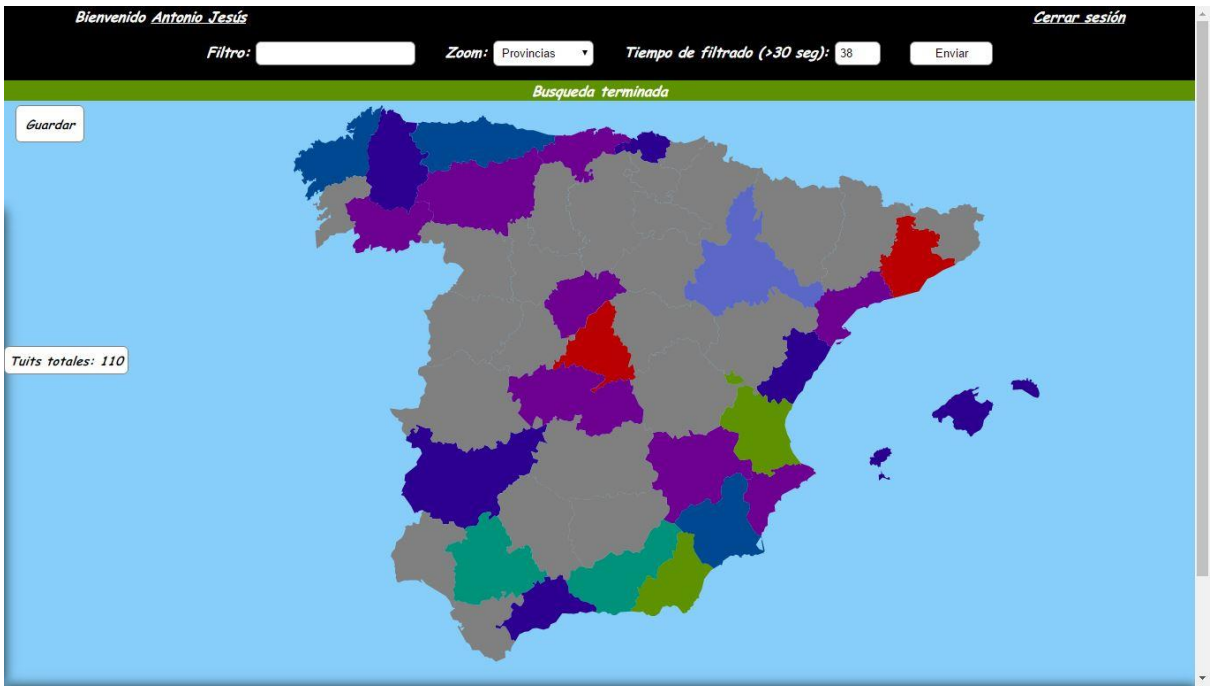


Ilustración 42: Búsqueda finalizada

Cuando la búsqueda haya finalizado, se podrá pulsar en “Tuits totales” y se mostrará un panel con un listado de las regiones que están pintadas en el mapa.

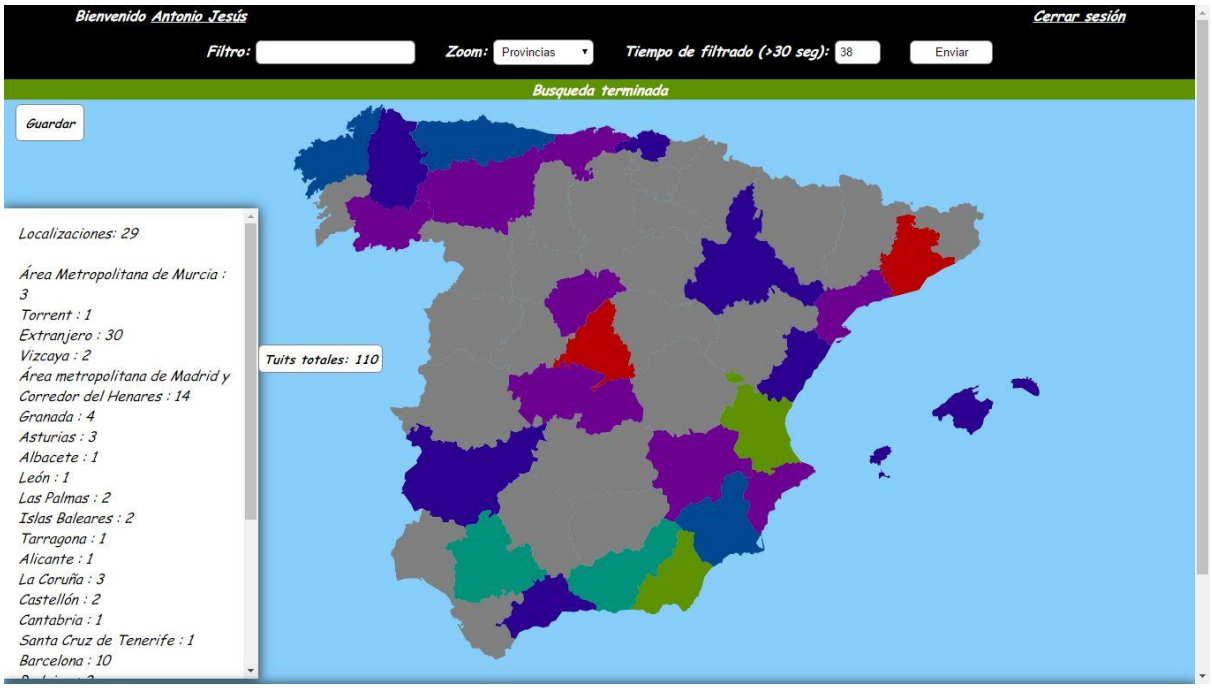


Ilustración 43: Ver localizaciones

Además, se podrá pulsar en cada región del mapa y se mostrará un panel con los *tuits* que se han producido en esa región. Los *tuits* se mostrarán de forma que se pueda ver el mensaje del *tuit* y el usuario que ha escrito ese *tuit*. Si pinchamos en el usuario, se nos redirigirá a la pantalla de perfil de Twitter de ese usuario.

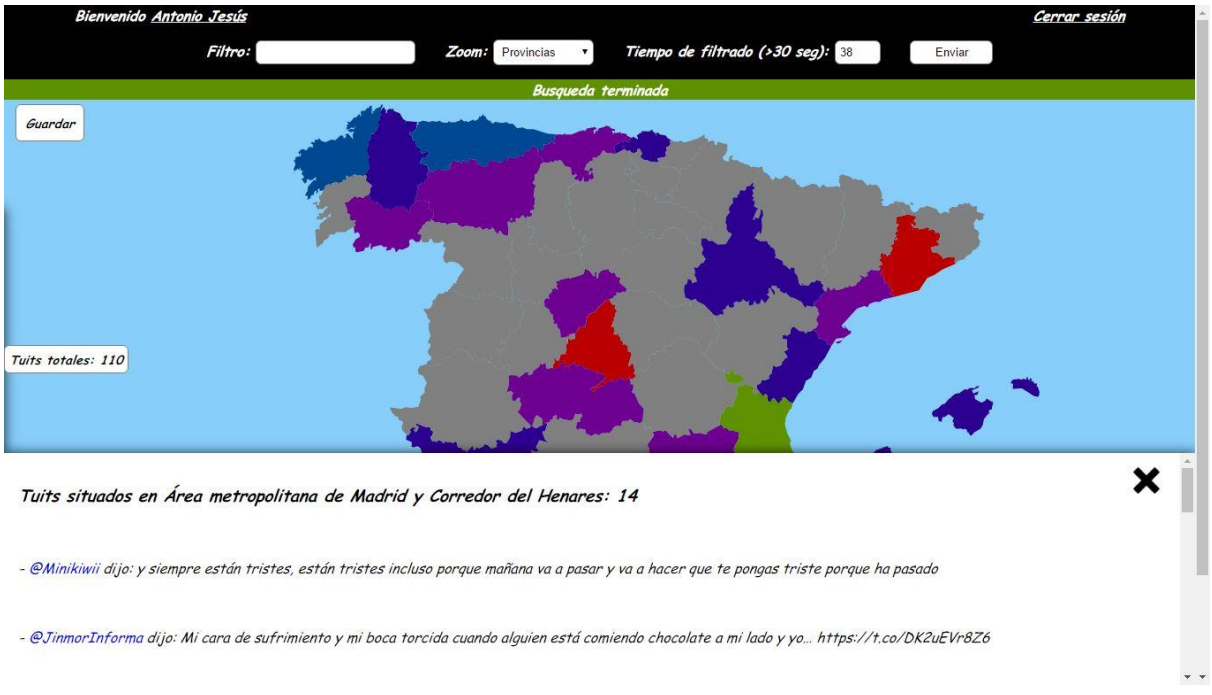


Ilustración 44: Ver tuits

Otra función presente en esta pantalla, es la de guardar el resultado obtenido de la búsqueda pinchando en el botón “Guardar”.

La última pantalla a la que se puede acceder es la de perfil de usuario, y se accede pinchando en el nombre del usuario que se encuentra en la esquina superior izquierda.

Ilustración 45: Ver perfil

En esta pantalla se muestran los datos del usuario, una tabla con las búsquedas que ha guardado el usuario y la posibilidad de modificar la contraseña de acceso.

Para cambiar la contraseña, simplemente hay que especificar la contraseña actual y la contraseña nueva y la contraseña quedará modificada.

Ilustración 46: Contraseña modificada

C. Manual de desarrollador

En este manual, se especifican los archivos que son modificables del proyecto por el desarrollador con el fin de mejorar la aplicación.

a. Establecer base de datos

Para establecer la url y las credenciales de a base de datos que se va a utilizar una vez desplegado el proyecto, habrá que modificar el archivo **hibernate.cfg.xml** (punto 8.2.2 de la documentación).

Este archivo se puede modificar incluso con el proyecto ya desplegado en el servidor correspondiente.

b. Establecer TwitterKeys

Las TwitterKeys son unos códigos que se obtienen de la web de twitter [15] y que permite usar el API de la red social.

Para hacer funcionar la aplicación, se necesita establecer las 4 *keys* que proporciona Twitter y se pueden modificar en el archivo ***twitter.properties*** del proyecto.

c. Añadir funcionalidades

Se pueden añadir dos tipos de funcionalidades, funcionalidades relacionadas con la base de datos, o relacionadas con la topología de *Storm.*

- Para añadir funcionalidades a la base de datos como pueden ser búsquedas más personalizadas o incluso más tipos de datos que se traducen en más tablas, se deben crear entidades e indicar las anotaciones correspondientes, crear los archivos *DAO* para cada entidad y posteriormente indicar las entidades mapeadas en el archivo de configuración de hibernate. Esto se puede ver con más detalle en el punto 8.2 del documento.

- Por otra parte, si se quiere modificar la topología de *Storm* lo recomendable es crear nuevas clases que contengan los *bolts* que realizarán las tareas correspondientes y realizar la pertinente conexión en el método que lanza la topología. No sería necesario retocar el *Spout* puesto que éste se encarga solo de recoger la información que se produce en Twitter y no de manipularla.